Universidad Carlos III de Madrid
Escuela Politécnica Superior
Grado en Ingeniería Informática
Mención de computación

Generación de contenido procedural en videojuegos basados en Unity

Trabajo de Fin de Grado

por

Daniel Jerez Garrido

5 de Julio de 2017

Tutor:
Yago Sáez Achaerandio

TRABAJO DE FIN DE GRADO

GENERACIÓN DE CONTENIDO PROCEDURAL EN VIDEOJUEGOS BASADOS EN UNITY

Autor: DANIEL JEREZ GARRIDO
Tutor: YAGO SÁEZ ACHAERANDIO

TRIBUNAL

Presidente: **Maria Belén Ruiz Mezcua**
Secretario: **Andrea Bellucci**
Vocal: **Luis Enrique Moreno Lorente**

Tras el acto de defensa y lectura el día 5 de julio de 2017 en la **Escuela Politécnica Superior** de la **Universidad Carlos III de Madrid** (Leganés), el tribunal le otorga la siguiente **CALIFICACIÓN**:

Agradecimientos

Para todas las personas que me apoyaron en este proyecto, en especial a mi hermana ***Sara****, que me supo animar en los momentos más difíciles, sin ella esto no sería posible.*
A mi amigo y compañero ***Mario Gonzalo****, que me acompañó, ayudó y aconsejó a lo largo de la carrera y a* ***Rafa del Riego****, desarrollador indie, que aunque le conozco poco me ofreció y enseñó consejos en Unity que me fueron de gran utilidad.* ***Gracias por todo.***

Dirección
Universidad Carlos III de Madrid
Escuela Politécnica Superior
Avda. de la Universidad
28911 Leganés (Madrid) - SPAIN

Por favor, cita este trabajo como:
Daniel Jerez Garrido: *Generación de contenido procedural en videojuegos basados en Unity*, TFG, Universidad Carlos III de Madrid, 2017.

Resumen

Este proyecto se basa en el desarrollo de un videojuego de plataformas en 2 dimensiones totalmente funcional para las plataformas Windows, Linux, Mac Os X y Android a través del motor gráfico Unity.

Se realizan técnicas de construcción de escenarios automatizados a través de carga de ficheros JSON editados por el usuario. Los ficheros contienen toda la información que el juego necesita para la puesta en escena (códigos de bloques, enemigos, fondo, música, inicio del jugador y meta). El nivel se estructura en una matriz bidimensional codificada que el juego recorre y va instanciando cada elemento con sus respectivas propiedades programadas.
Además, se estudian técnicas de generación procedural como alternativa a la creación manual de los niveles mencionados.

Palabras clave

Videojuego ; Plataformas ; Unity ; JSON ; Contenido Procedural

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se introduce el contexto que enmarca al proyecto *Kure World*. Consiste en el desarrollo de un videojuego totalmente funcional del género de plataformas de scroll lateral en 2 dimensiones con cámara ortográfica utilizando para su desarrollo el motor gráfico Unity y el lenguaje de programación C#. Se han seguido dos líneas de trabajo:

- Desarrollo y funcionalidad del videojuego. Aprendiendo el uso de la herramienta Unity desde cero con el fin de conocer las fases de desarrollo de un proyecto software y la construcción de un juego que cumpla unas mecánicas y un objetivo, además de una interfaz atractiva para el usuario.

- Investigación e implementación de métodos de carga de ficheros para la automatización en la construcción de niveles que siguen una jerarquía y codificación planteada, además de un estudio de generación procedural que permita la generación automática de estos.

1.1. Estructura de la memoria

La memoria sigue una jerarquía guiada, hablando de forma general al principio y profundizando en lo específico poco a poco. El capítulo actual corresponde con la introducción, motivación y objetivo principal del trabajo. El capítulo 2 contiene el estado del arte que abarca la temática del proyecto. El capítulo 3 se basa en la explicación del motor Unity 3D y los elementos utilizados en el desarrollo. El capítulo 4 es el núcleo que explica el análisis, diseño y posterior implementación que se ha seguido a lo largo del proyecto. La parte final de la memoria consta del capítulo 5, conclusión general, capítulo 6 para trabajos futuros asociados a este proyecto y capítulo 7 que abarca todo el tema de gestión, presupuesto y un manual de usuario. El último capitulo es la inclusión de un resumen en inglés correspondiente al plan 2011 de ingeniería informática.

1.2. Motivación

La principal idea y mecánica del proyecto surgió en base a la experimentación de meterse en el mundo independiente del desarrollo de un videojuego. La experiencia personal era solo como jugador, pero siempre me llamó la atención crear un juego propio y personal que pudiese ser disfrutado y que genere una crítica constructiva en una comunidad.

La mayoría de los juegos de plataformas que existen hoy en día congregan a un conjunto de expertos diseñadores que adaptan la experiencia de juego dentro de sus objetivos y mecánicas jugables, proporcionando un concepto que se basa en su mayoría en algo lineal y totalmente planificado.

La idea original del proyecto era poder compartir y publicar escenarios y niveles que cualquier usuario con un simple editor de texto pudiera construir, sin necesidad de tener estos conceptos previos de diseño. Este planteamiento estaría ligado a la concatenación de niveles para generar una experiencia personalizada dentro de una misma plataforma jugable. Como parte alternativa a este diseño me pregunté si en vez de poner a un usuario como diseñador de niveles, este podría ser sustituido por un algoritmo capaz de crear un mundo totalmente nuevo en cada partida proponiendo retos interesantes al jugador.

Dos de los juegos referentes que están presentes en esta idea de concepto son, por un lado, el famoso fontanero *Super Mario Bros*, que dentro de las plataformas en scroll lateral 2D es de los más famosos. Desde un principio los sprites que se utilizaron para conceptualizar el juego fueron los de *Super Mario World* para la consolas *SNES*. Por otro lado, el otro juego es *The binding of Isaac*, no tan reconocido mundialmente, pero sí valorado dentro de la comunidad indie como una obra maestra. El juego se basa en mazmorras aleatorias generadas de forma procedural con pisos que constan de incrementos de dificultad y una evolución del personaje basada en el concepto roguelike totalmente aleatoria.

Teniendo claro estas ideas y juegos, la principal motivación era crear un producto agradable visualmente, jugable y entretenido que mezcle una ilusión por los videojuegos y un futuro dentro de estos.

1.3. Objetivos del trabajo

Los principales objetivos del proyecto se pueden definir en los siguientes:

- Realizar un videojuego de plataformas en 2D con la herramienta Unity y hacerlo totalmente funcional para las plataformas Windows, Mac OS X, Linux y Android.
- Conseguir realizar una carga de niveles contenidos en ficheros JSON que contengan una matriz con los elementos codificados además de otros atributos como el fondo, la música, el número de vidas, el autor del nivel y como parte opcional el siguiente nivel.
- Diseñar una interfaz que permita manipular el videojuego, pudiendo elegir entre el listado de niveles disponibles en ese momento.
- Realizar un estudio de una posible generación procedural de niveles que sea compatible con esta estructura de juego.

Capítulo 2

Estado del arte

El videojuego se creó como una herramienta de entretenimiento, que con el paso del tiempo ha ido generando más y más público interesado. A día de hoy es uno de los mercados más importantes y fructíferos, pero para conocer un poco el concepto que mueve el videojuego, es importante saber lo que es un motor gráfico, una breve introducción a la historia hasta situarnos en el momento actual, y algo más concreto, como es el género de plataformas dentro de otros muchos y las posibles herramientas de generación procedural aplicadas a este.

2.1. Motor de videojuegos

El núcleo central de un videojuego es su motor gráfico, plataforma encargada de representar el juego a través de un conjunto de rutinas de programación y múltiples opciones de diseño y creación. Estas herramientas son capaces de realizar cálculos geométricos y de físicas, que permiten recrear el mundo diseñado como si fuera un ecosistema.

Por lo general, las limitaciones que tiene el motor están acotadas por el videojuego desarrollado dando lugar a motores exclusivos para determinados juegos. Dentro de las capacidades más importantes, el motor gráfico funciona con la estructura de un lenguaje de programación (en el caso de Unity el lenguaje es C#) donde cada elemento que compone el juego tiene un script asociado con la programación pertinente o es un componente diseñado por la comunidad, siendo estos últimos de pago o de uso libre. Los más populares son los de carga, animación, detracción de colisiones, la física, la interfaz gráfica o determinadas herramientas de inteligencia artificial. El juego en su mayoría está compuesto por modelados, texturas, programación y comportamiento ante colisiones que permiten la interacción con el entorno. [3].

Las características principales de las que dispone un motor gráfico actual son:

- Creación y visualización de elementos 3D y/o 2D con físicas de comportamiento, iluminación, sombreados, reflejos en tiempo real y texturas.
- Integración de elementos sonoros, musicales, de interfaz, acceso, controladores y renderización.
- Gestión de red para componentes multijugador y otros componentes creados o modificados por la comunidad.

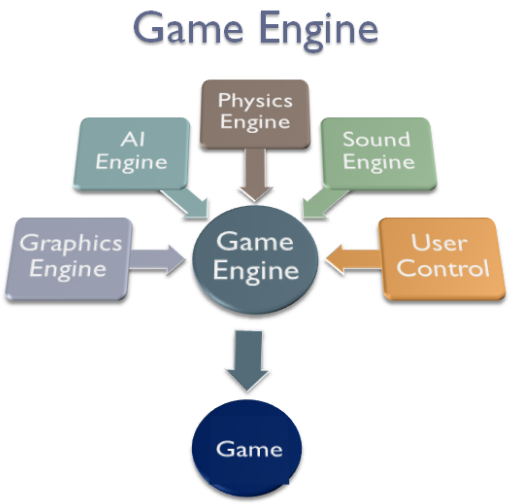

Figura 2.1: *Estructura de un motor gráfico (Game Engine).*
Fuente: `http://factions.pidbaq.com/whats-a-good-game-engine/`

Los motores suelen ser herramientas de pago, pero también existe un gran mercado gratuito o intermedio, con presupuestos para distintos tipos de desarrolladores. Dentro de estos últimos mencionados se encuentra Unity 3D, el motor utilizado para el desarrollo de este proyecto.

Figura 2.2: *Motores gráficos más populares.*

2.2. Historia de los videojuegos

Los videojuegos tienen su origen en la década de los años 40. Después de finalizar la Segunda Guerra Mundial, las principales potencias se embarcaron en una carrera tecnológica donde se construyeron las primeras computadoras programables como ENIAC (pesaba unas 27 toneladas y ocupaba una superficie de 167 m^2).

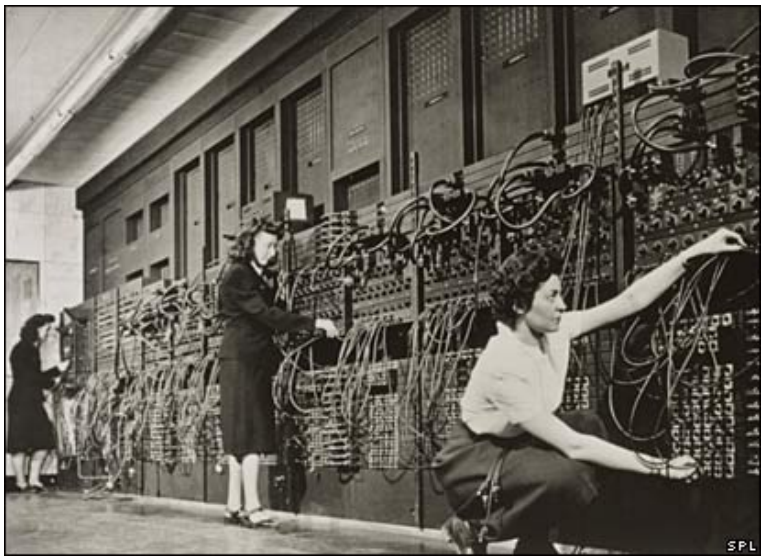

Figura 2.3: *Foto archivo de Eniac.*

Los primeros experimentos y pruebas académicas designados como videojuegos eran básicamente ensayos científicos y físicos. La verdadera industria no empezó hasta los años 70, donde estos empezaron a comercializarse y se marcó una etapa de transformación y continua evolución del concepto que con el paso del tiempo se ha convertido en todo un fenómeno cultural logrando un estatus de medio artístico en algunos países [7].

En el año 1958, William Higginbotham utilizando un programa de cálculo de trayectorias y un osciloscopio creó un simulador de tenis de mesa llamado Tennis fo Two (tenis para dos), que servía de entretenimiento en el Brookhaven National Laboratory a sus visitantes. El juego se considera el primer juego con capacidad de dos jugadores. Unos años más tarde, Steve Russel [8], estudiante del Instituto de Tecnología de Massachussets y con un desarrollo de seis meses en su computadora, creó Spacewar, un juego basado en gráficos vectoriales que funcionaba sobre un PDP-1 (Programmed Data Processor-1) donde dos jugadores controlaban dos naves espaciales que se enfrentaban entre ellas. El videojuego no fue muy reconocido fuera del ámbito universitario, pero tuvo mucho éxito entre sus seguidores.

En el año 1966 un pequeño grupo formado por Ralph Baer, Albert Maricon y Ted Dabney [1] empezó el desarrollo de un videojuego que se tituló Fox and Hounds siendo el precursor del videojuego doméstico. El proyecto evolucionó y se convirtió en Magnavox Odyssey, lanzado en 1972, el juego se conectaba a un televisor y permitía jugar a varios juegos pregrabados [2].

La eclosión de los videojuegos (1970-1979)

Una mención importante en el inicio del videojuego es para Nolan Bushnell que en 1971 empezó a comercializar Computer Space, una versión adaptada y modificada de Space War que anteriormente se había publicado en una versión recreativa titulada Galaxy War a principios de los 70 en la universidad de Standford.

Donde verdaderamente se creó una ascensión del videojuego fue con la salida de la recreativa del videojuego Pong (anteriormente llamado Tennis for Two y desarrolado por Higginbotham), sistema adaptado y diseñado por el ingeniero Allan Alcorn para Atari, compañía recién fundada por Nolan Bushnell.

Fue presentado en 1972 y supuso la base del videojuego como una industria. En los siguientes años la evolución técnica fue avanzando, desde la inclusión de chips de memoria y procesadores más potentes que permitieron desarrollar juegos como Space Invaders (Taito) o Asteroids (Atari) que fueron publicados salones recreativos [2].

La década de los 8 bits (1980-1989)

En la década de los 80 se experimentó un fuerte crecimiento en el sector de los videojuegos donde las máquinas recreativas eran cada vez más populares y ver videoconsolas en hogares era cada vez más normal. Las consolas más destacadas eran Oddyssey 2 (Phillips), Intellivision (Mattel), Colecovision (Coleco), Atari 5200, Commodore 64, Turbografx (NEC) y los juegos de las recreativas que más triunfaban eran Pacman (Namco), Battle Zone (Atari), Pole Position (Namco), Tron (Midway) o Zaxxon (Sega).

Tras un gran mercado a principio de los 80, sobre el año 1983 comenzó la denominada crisis del videojuego que afectó principalmente a países como Estados Unidos y Canadá y abarcó hasta el año 1985.

En 1983 una compañía japonesa llamada Nintendo apostó por el lanzamiento de su consola Famicon, llamada en occidente NES (Nintendo Entertainment System) la cual tuvo un gran éxito en todo el mundo. Por otro lado, Europa se centró en los microrordenadores como Spectrum y Commodore.

Tras la salida de la crisis, los norteamericanos decidieron adoptar la NES como su principal sistema de videojuegos lo que generó un gran interés en la creación de sistemas domésticos como Master System (Sega), el Amiga (Commodore) y el 7800 (Atari) con títulos destacados como Tetris.

A finales de los 80 se empezaron a ver las primeras consolas de 16 bits, como Mega Drive de Sega y los microprocesadores poco a poco se iban sustituyendo por arquitecturas IBM. En 1985 Nintendo sacó al mercado el juego Super Mario Bros, considerado uno de los juegos más importantes de la historia y que supuso un punto de inflexión en la industria del desarrollo, introduciendo mecánicas que iban mas allá de conseguir una puntación alta en una serie de niveles repetidos. Super Mario Bros tenía un objetivo, un final y conseguía tener variedad de escenarios y elementos, lo que obligó a otras compañías tener que emular su estilo.
Los videojuegos de recreativas destacados en esta etapa fueron Defender, Rally-X, Dig Dug, Bubble Bobble, Gauntlet, Out Run o Shinobi destacando a Japón como la mayor productora de estos.

Un mercado que también creció en esta etapa fue el de las consolas y videojuegos portátiles. Los primeros precursores aparecieron en los 70 con la consola Mattel, pero fue la adaptación de recreativas adaptadas por Coleco y minijuegos de Nintendo como Game & Watch lo que permitió su auge. La evolución de este concepto llegó en 1989 con el lanzamiento de la Game Boy por parte de Nintendo [2].

Figura 2.4: *Consolas de los 80 (Famicom, Mega Drive, Coleco y la portátil Game Boy)*

La revolución de las 3D (1990-1999)

La «generación de 16 bits» de principios de los 90 supuso una gran competición y salto técnico. En su mayoría estaba compuesta por Mega Drive, la Super Nintendo Entertainment de Nintendo, la PC Engine de NEC, conocida como Turbografx en occidente y la CPS Changer de (Capcom).

La inclusión de tecnología y gráficos 3D fue por parte de consolas como Nintendo 64, Playstation y Sega Saturn que tuvieron una gran acogida y éxito de ventas. Otra consola a destacar fue Neo GEO (SNK) que igualaba en prestaciones a un arcade recreativo pero tenía un coste realmente alto, que la impedía llegar a muchos hogares [2].

El comienzo del nuevo siglo (Desde el 2000)

La compañía japonesa Sony en el año 2000 lanzó al mercado PlayStation 2, mientras que Sega lo hizo con Dreamcast, una consola con características similares. Microsfot decidió entrar en la industria creando la Xbox en el año 2001, mientras que Nintendo lo hacía con Gamecube y en el mercado portátil con GameBoy Advance. Frente a la gran acogida que tuvo PlayStation 2, en 2002 la compañía Sega decidió abandonar la producción de hardware y centrarse en el desarrollo software.

Los ordenadores personales también se hicieron un hueco en la industria del videojuego, siendo la plataforma más cara pero la que permitía mayor flexibilidad, como añadir componentes que mejoran la experiencia de juego (tarjetas gráficas o de sonido, periféricos como mandos y accesorios) y permitiendo instalar parches oficiales o realizados por la comunidad para arreglar fallos o incluir mejoras al juego [2]. En la actualidad las consolas y ordenadores son cada vez más potentes, pero el concepto es siempre el mismo. Algunas de las consolas destacadas son Xbox One, PlayStation 4 y Nintendo Switch.

Figura 2.5: *Consolas de sobremesa de la generación 2000 (Gamecube, Xbox, Playstation 2)*

2.3. Videojuegos de plataformas

El género de plataformas se caracteriza por controlar a un personaje a lo largo de un nivel lleno de obstáculos, plataformas y enemigos con distintas mecánicas y cuyo objetivo principal es completarlo cumpliendo unos requisitos (tiempo, vidas, meta, etc). Las principales mecánicas de las que dispone el género son caminar, saltar, correr, escalar, disparar, nadar, volar, etc. La cámara de los plataformas 2D suele ser ortográfica y su desplazamiento es en scroll lateral (derecha e izquierda) aunque algunos títulos incluyen verticalidad (arriba y abajo). Los primeros plataformas 2D surgen en la década de los 80 y actualmente es un género muy popular.

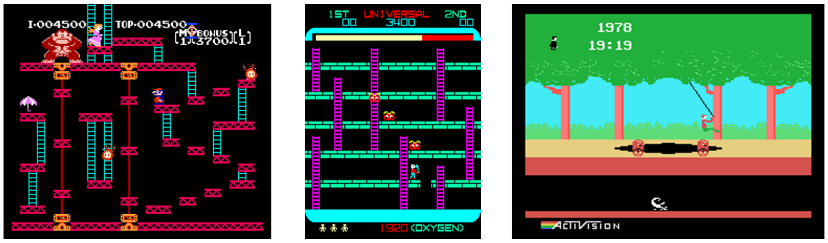

Figura 2.6: *Donkey Kong / Space Panic / Pitfall.*

La localización de su origen es en las máquinas Arcade, en juegos como Space Panic (1980) y Donkey Kong (1981) entre otros, pero el género no se definió hasta la salida de Pitfall (1982) para la consola Atari 2600. La compañía Nintendo fue la que perfeccionó el género con el título Super Mario Bros (1985) para la consola NES, algunas de las características destacables fue la inclusión de niveles por mundos, objetos que permitían cambiar al personaje (denominados powers up), contadores de vida y tiempo y una gran variedad de mecánicas. Otros títulos famosos de la época fueron Metroid (1986), Castlevania (1989) y Prince of Persia (1989). En la mitad de la década de los 90 fueron apareciendo títulos de 16 bits como Sonic the Hedgehog o Super Mario World, este último sirvió de inspiración para este proyecto. Algunas mejoras posteriores del género fue la inclusión de un sistema pre-renderizado de gráficos en títulos como Donkey Kong Country o la llegada del 3D con consolas como Nintendo 64 y títulos como Super Mario 64 donde el jugador podía explorar libremente un mundo tridimensional [6].

Figura 2.7: *Evolución de Mario (1985, 1990, 1996)*

La generación del 2000 siguió con la fórmula de los 32 y 64 bits. Los títulos más destacados son Super Monkey Ball, Crash Bandicoot, Jak and Daxter, Ratchet & Clank y Spyro the Dragon que pulieron el género y dejaron personajes famosos en la historia. En la actualidad este género tiene su auge en el 2D pues la salida de pequeños grupos de desarrolladores indies se centran en él por su menor coste de desarrollo, la posibilidad de ofrecer un arte elaborado y evitar problemas de cámara que muchos plataformas 3D no han conseguido arreglar, la mención más destacable de estos trabajos sería Super Meet Boy (2010) desarrollado por Edmund McMillen y Tommy Refenes.

Figura 2.8: *Personajes famosos del género plataformas. (Mario, Sonic, Spyro y Crash)*

2.4. Generación procedural en juegos

La generación de contenido procedural es una técnica que data del año 1973 y consiste en la generación y construcción automatizada de escenarios o elementos sin necesidad de un diseño manual y con la capacidad de un uso escaso de memoria al generarse sobre la marcha. Otra característica es la aleatoriedad y variedad de las generaciones y elementos.

Se empezó a utilizar en juegos como Beneath Apple Manor que consistía en la construcción de mazmorras con habitaciones, monstruos y tesoros de forma aleatoria. Es una técnica que se utilizó mucho debido a las bajos recursos que disponían las consolas y computadoras de la época, pero que en la actualidad se ha recuperado por las capacidades de diseño variado que permite si el algoritmo está bien construido, permitiendo que los títulos sean más rejugables y el jugador tenga una experiencia nueva en cada partida. El género donde más destaca es en Roguelike en grupos de desarrollo independiente. Algunos títulos destacados son Minecraft, Spelunky, The binding of Isaac o Diablo [9].

Capítulo 3

Unity

La historia nos cuenta hasta dónde ha llegado un videojuego, pero si somos capaces de adentrarnos en su núcleo, llegamos a la unificación de un motor gráfico compuesto de miles de componentes que funcionan a la vez en una determinada escena que ve el usuario. No siempre se utilizan todas las capacidades del motor, pero es necesario conocer el uso de las más importantes y necesarias que aparecen en la mayoría de estos e indirectamente se complementan.

3.1. Introducción al motor

El videojuego desarrollado para este proyecto, llamado *Kure World*, ha utilizado el motor gráfico Unity en su versión 5.6.1f1 (64-bit) personal. Unity es un motor de videojuegos multiplataforma creado por Unity Technologies disponible para Microsoft Windows y OS X.

Fue mostrado por primera vez en una conferencia de Apple dada su exclusividad con la plataforma OS X y posteriormente lanzado en 2005 y debido al gran éxito que tuvo también se desarrolló en Windows. En el año 2010 salió la versión 3 que incluyó gran variedad de herramientas tanto para profesionales como grupos pequeños que no se podían permitir altos costes. La versión actual es Unity 5 y fue lanzada en 2015 e incluyo soporte para Direct X 11/12 y compatibilidad con las distribuciones de Linux. A continuación, se desglosan los componentes más importantes del motor que han sido utilizados en el proyecto, toda esta información ha sido sacada del manual oficial de Unity disponible en su página web.

Figura 3.1: *Logo del motor gráfico Unity.*

3.2. Prefabs

Un prefab es un tipo de asset (representación de un objeto que se utiliza en los proyectos) que permite almacenar un objeto con determinadas componentes y propiedades que lo identifican como una plantilla que se puede instanciar en la escena tantas veces sea necesario. La modificación del prefab también altera el comportamiento de las instancias de este, aunque estas pueden modificarse individualmente y crear nuevos prefabs [11].

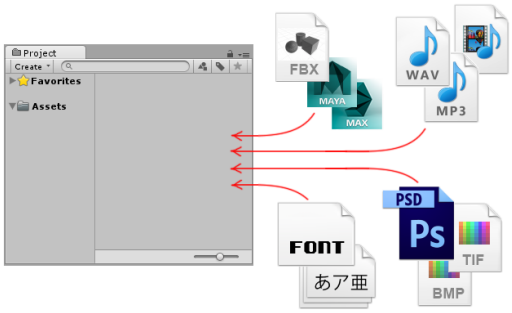

Figura 3.2: *Ejemplo de interfaz y ficheros que son Assets.*

3.3. Colliders

Los collider son componentes invisibles dentro de objetos que permiten el cálculo de colisiones físicas entre otros objetos. Se definen a través de una malla que abarca la zona donde se requiere el cálculo de la colisión.

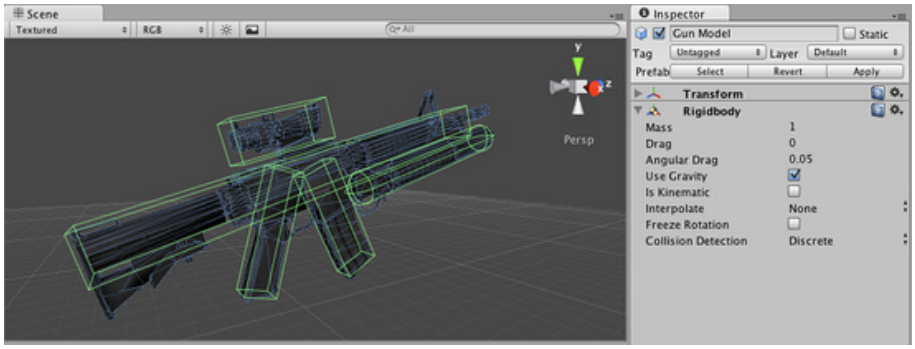

Figura 3.3: *Objeto 3D compuesto con Box Colliders.*

Los colliders usados en elementos 3D son Box Collider, Sphere Collider y Capsule Collider. En 2D, se utilizan Box Collider 2D y Circle Collider 2D. Los objetos pueden tener conjuntos de colliders para determinadas zonas de este [11].

3.4. Físicas

Las físicas son una parte fundamental dentro del diseño del videojuego pues permiten representar un comportamiento físico convincente en esa realidad. Los objetos deben ser alterados de forma adecuada por las colisiones, la gravedad y las fuerzas. Unity proporciona una herramienta de físicas que ajusta los parámetros establecidos y los simula de manera pasiva o realista (No se alteran por sí solos sino por la interacción de agentes externos). Los scripts permiten añadir mecánicas de físicas a los objetos como por ejemplo de vehículos, tela, agua, etc [11]. En este proyecto en concreto, se han utilizado Rigibodys 2D, que son un componente que permite la inclusión de control y físicas 2D (ejes cartesianos) a Sprites.

3.5. Cámara

Una cámara es un objeto que define una vista. Las escenas en Unity se espectan con cámaras que representan una zona del espacio tridimensional acotada por unas dimensiones y ángulos en dos dimensiones, que se encarga de aplanar el conjunto de vectores para mostrarlos por pantalla.

Se pueden tener una o varias cámaras pero solo una de ellas puede estar activa a la vez. La cámara usada en el juego es de proyección ortográfica, representa elementos geométricos en un plano mediante protección ortogonal. El uso de esta técnica permite mantener proporciones en la escena con ángulos rectos [11].

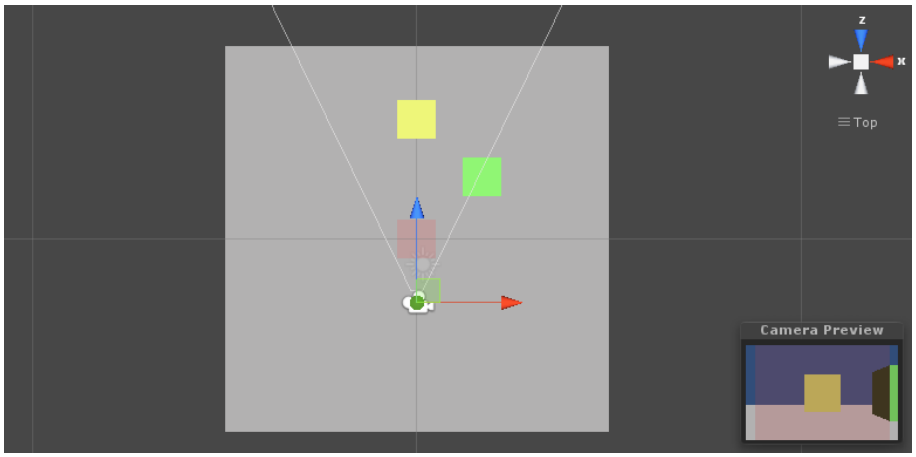

Figura 3.4: *Ejemplo de interfaz de uso de cámara.*
Fuente: `http://unityjs.blogspot.com.es/2013/04/ocultando-objetos-que-se-interponen-en.html`

3.6. Controladores de Personaje

El controlador permite la interacción entre el jugador y el personaje. Este controlador ha de ser lo más preciso posible y debe interactuar bien con el resto de elementos que generan una colisión, como obstáculos, enemigos u objetos, o determinados cambios en la física como puede ser saltar, desplazarse o recibir daño.

Para este proyecto se ha utilizado la librería disponible GetAxis (en Android touchController que muestra un joystick y botones táctiles en pantalla) que permite detectar los cambios de velocidad en los ejes verticales y horizontales, que indican un desplazamiento negativo o positivo en los ejes. Esta interpretación se representa en movimiento en el personaje y es lo que permite al jugador moverse por el escenario. Otro método de entrada es la captación de botones pulsados que permiten realizar determinadas acciones como saltar, disparar o correr.

Los periféricos asociados al controlador son teclado, ratón, mando y en dispositivos Android la pantalla táctil. El componente que permite la comunicación del objeto con el controlador es el Rigidbody [11].

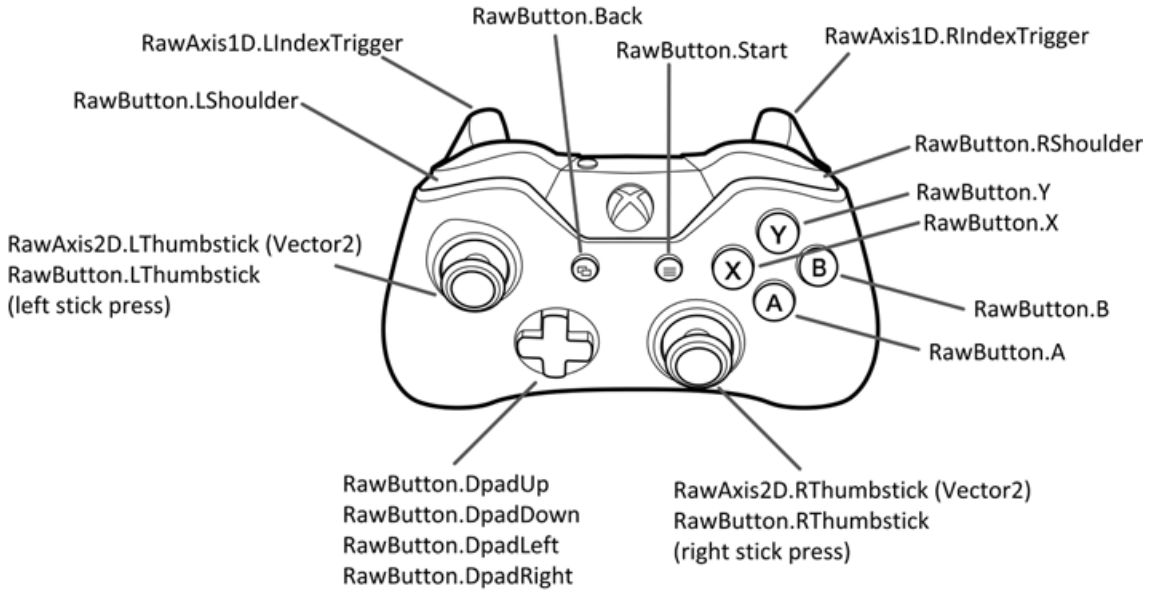

Figura 3.5: *Ejemplo de controlador - Posicionamiento de botones en mando de Xbox 360.*
Fuente: `https://developer.oculus.com/`

Capítulo 4

Análisis, diseño e implementación

Las claves necesarias para construir un buen concepto estructurado son el análisis, el diseño y la implementación. Este capítulo divide estos apartados y los documenta de forma específica.

4.1. Análisis

La sección esta divida en casos de uso, que muestran las implicaciones a las que se somete el juego, requisitos, que especifican las capacidades y, diagramas, que muestran el flujo operativo de forma visual.

4.1.1. Casos de uso

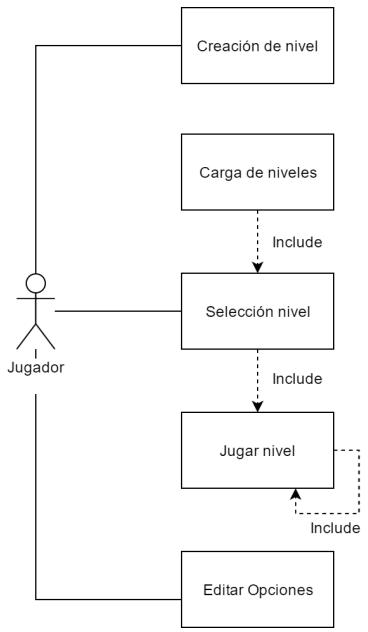

Figura 4.1: *Diagrama de casos de uso del sistema.*

Tabla 4.1: Caso de uso - Creación de nivel.

Identificador	**CU01**
Nombre	Creación de nivel
Actor	Jugador
Descripción	El jugador podrá crear un nivel con cualquier editor de texto. Este vendrá en una estructura JSON que incluirá los parámetros vidas, tiempo, fondo, música, autor, siguiente nivel, y matriz del nivel. En propio juego generará 3 niveles de ejemplo para ayudar al jugador a entender la estructura.
Precondiciones	Abrir un editor de texto con un documento vacío o editar un nivel por defecto dentro de la carpeta autogenerada *Levels.*
Postcondiciones	Documento JSON creado y almacenado en la carpeta *Levels.*

Tabla 4.2: Caso de uso - Carga de niveles.

Identificador	**CU02**
Nombre	Carga de niveles
Actor	Aplicación
Descripción	La aplicación listará los niveles disponibles en la carpeta raíz *Levels* cuyo formato sea *.map* y se mostrará por pantalla en un componente *list* desplegable.
Precondiciones	Tener algún nivel disponible en la carpeta *Levels* o finalizar un nivel con parámetro que apunte a otro nivel a cargar.
Postcondiciones	El jugador puede elegir entre los niveles.

Tabla 4.3: Caso de uso - Selección nivel.

Identificador	**CU03**
Nombre	Selección nivel
Actor	Jugador
Descripción	El jugador podrá elegir entre los niveles listados en la carga de niveles.
Precondiciones	Elegir un nivel correctamente construido y almacenado.
Postcondiciones	La aplicación construye el escenario y los elementos.

Tabla 4.4: Caso de uso - Jugar nivel.

Identificador	**CU04**
Nombre	Jugar nivel
Actor	Jugador
Descripción	La aplicación carga el nivel y el jugador puede jugar en él.
Precondiciones	Elegir un nivel o terminar un nivel con continuación.
Postcondiciones	Si el nivel no tiene continuación el juego se pausa y salen opciones de salir y reintentar nivel.

Tabla 4.5: Caso de uso - Editar Opciones.

Identificador	**CU05**
Nombre	Editar Opciones
Actor	Jugador
Descripción	El jugador puede editar opciones de volumen, muteo y disposición del *hud* desde la interfaz principal.
Precondiciones	Tener la aplicación ejecutando.
Postcondiciones	Se aplica la nueva configuración elegida.

4.1.2. Requisitos

Los requisitos se dividen en usuario, funcionales y no funcionales. Permiten conocer las capacidades planteadas en el proyecto, su prioridad y la necesidad de esta.

4.1.2.1. Requisitos de usuario

Este tipo de requisito se centra en las posibilidades y circunstancias que tendrá un usuario al enfrentarse al producto.

Tabla 4.6: Requisito de usuario - Ejecutable.

Identificador	**RU01**
Nombre	Ejecutable
Descripción	El videojuego está compilado para las plataformas Windows, Mac OS X, Linux y Android en sus respectivos archivos ejecutables.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.7: Requisito de usuario - Selección de nivel.

Identificador	**RU02**
Nombre	Selección de nivel
Descripción	El jugador puede elegir un nivel en concreto mostrado en una lista desplegable. Estos niveles se almacenarán en una carpeta raíz llamada Levels.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.8: Requisito de usuario - Jugar nivel.

Identificador	**RU03**
Nombre	Jugar nivel
Descripción	El jugador puede jugar niveles. El objetivo es acabarlos en el menor tiempo posible con la máxima puntuación que se divide en enemigos eliminados y monedas recogidas.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.9: Requisito de usuario - Opciones.

Identificador	**RU04**
Nombre	Opciones
Descripción	En la interfaz principal se muestran las opciones de modificar sonido, activar/desactivar música y mostrar/ocultar el hud.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.10: Requisito de usuario - Control de personaje.

Identificador	**RU05**
Nombre	Control de personaje
Descripción	El personaje se podrá controlar con teclado o mando. Las acciones disponibles son saltar, andar, correr, agacharse y disparar.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.11: Requisito de usuario - Pausar.

Identificador	**RU06**
Nombre	Pausar
Descripción	En cualquier momento de la partida se puede pausar el juego. Las opciones de pausa son reintentar el nivel, salir, ocultar el hud y modificar los niveles de sonido.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.12: Requisito de usuario - Recoger monedas.

Identificador	**RU07**
Nombre	Recoger monedas
Descripción	El jugador puede recoger monedas, estas aparecen repartidas por el escenario o ocultas en bloques. Cada moneda recogida incrementa el contador de monedas en uno.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.13: Requisito de usuario - Enemigos.

Identificador	**RU08**
Nombre	Enemigos
Descripción	El jugador puede eliminar algunos enemigos. Los enemigos aparecen repartidos por el escenario. Eliminar un enemigo incremente el contador de enemigos en uno.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.14: Requisito de usuario - Perder vida.

Identificador	**RU09**
Nombre	Perder vida
Descripción	El jugador puede perder vidas si, se acaba el tiempo, un enemigo toca al personaje sin ningún power up, el personaje se cae al vacío o sale fuera del escenario y un proyectil enemigo impacta en el personaje. Si el jugador se queda sin vidas el juego se termina.
Prioridad	Alta
Necesidad	Esencial

Tabla 4.15: Requisito de usuario - Terminar nivel.

Identificador	**RU10**
Nombre	Terminar nivel
Descripción	Si el jugador toca la bandera se termina el nivel. Si el nivel no tiene continuación se pausa la partida. Si el nivel si tiene continuación se carga directamente el siguiente nivel.
Prioridad	Alta
Necesidad	Esencial

4.1.2.2. Requisitos funcionales

Este tipo de requisito se encarga de especificar capacidades concretas dentro del proyecto.

Tabla 4.16: Requisito funcional - Tecla de pausa.

Identificador	**RF01**
Nombre	Tecla de pausa
Descripción	La tecla para pausar el juego es ESCAPE en teclado y START en mando.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.17: Requisito funcional - Teclas de movimiento.

Identificador	**RF02**
Nombre	Teclas de movimiento
Descripción	Las teclas de movimiento son WASD y direcciones en teclado, y Joystick izquierdo en mando.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.18: Requisito funcional - Tecla de salto.

Identificador	**RF03**
Nombre	Tecla de salto
Descripción	La tecla de salto es ESPACIO en teclado y Botón 1 en mando.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.19: Requisito funcional - Volumen.

Identificador	**RF04**
Nombre	Volumen
Descripción	En la interfaz principal o en el menú de pausa del nivel se puede elegir la intensidad del volumen en una escala 0 a 100.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.20: Requisito funcional - Activar/desactivar música.

Identificador	**RF05**
Nombre	Activar/desactivar música
Descripción	En la interfaz principal o en el menú de pausa del nivel se puede elegir activar o desactivar la música.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.21: Requisito funcional - Activar/desactivar HUD.

Identificador	**RF06**
Nombre	Activar/desactivar HUD
Descripción	En la interfaz principal o en el menú de pausa del nivel se puede elegir activar o desactivar el HUD.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.22: Requisito funcional - Ejecutar juego.

Identificador	**RF07**
Nombre	Ejecutar juego
Descripción	El juego se lanza con el ejecutable compilado del proyecto.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.23: Requisito funcional - Salir del juego.

Identificador	**RF08**
Nombre	Salir del juego
Descripción	Para salir del juego se puede pulsar el botón ESCAPE o el botón situado en la esquina superior derecha con una X.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.24: Requisito funcional - Listar niveles.

Identificador	**RF09**
Nombre	Listar niveles
Descripción	Se listan todos los niveles en formato .map disponibles en la carpeta raíz Levels. Si la carpeta no existe se crea y se generan 3 niveles de ejemplo.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.25: Requisito funcional - Recoger moneda.

Identificador	**RF10**
Nombre	Recoger moneda
Descripción	Se pueden recoger monedas en los niveles, esto incrementa el contador de monedas en 1 por cada una recogida.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.26: Requisito funcional - Eliminar enemigo.

Identificador	**RF11**
Nombre	Eliminar enemigo
Descripción	Se pueden eliminar enemigos en los niveles, esto incrementa el contador de enemigos en 1 por cada una eliminado.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.27: Requisito funcional - Movimiento enemigo.

Identificador	**RF12**
Nombre	Movimiento enemigo
Descripción	Los enemigos tienen movimientos automáticos horizontales. Cuando se topan con un obstáculo su sentido cambia. El movimiento del enemigo se activa cuando el jugador esté a una determinada distancia.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.28: Requisito funcional - Control cámara.

Identificador	**RF13**
Nombre	Control de cámara
Descripción	La cámara seguirá al jugador en todo momento con un scroll lateral.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.29: Requisito funcional - Volver al inicio.

Identificador	**RF14**
Nombre	Volver al inicio
Descripción	Se puede salir a la interfaz inicial pausando el juego y pulsando salir.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.30: Requisito funcional - Reintentar nivel.

Identificador	**RF15**
Nombre	Reintentar nivel
Descripción	Se puede reintentar el nivel con las estadísticas actuales desde el menú de pausa o al perder una vida.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.31: Requisito funcional - Disparar.

Identificador	**RF16**
Nombre	Disparar
Descripción	Se puede disparar si se consigue tocar una flor. El límite de disparos en pantalla es 3.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.32: Requisito funcional - Crecer.

Identificador	**RF17**
Nombre	Crecer
Descripción	Se puede crecer si se coge una seta o una flor. Esto permite recibir golpes de enemigos sin perder vida, solo la transformación.
Prioridad	Esencial
Necesidad	Alta

4.1.2.3. Requisitos no funcionales

Este tipo de requisito es el encargado de especificar criterios que pueden usarse para juzgar la operación de un sistema.

Tabla 4.33: Requisito no funcional - Formato de nivel.

Identificador	**RNF01**
Nombre	Formato de nivel
Descripción	Los niveles están en formato .map. Estos son modificables con cualquier editor de texto. Siguen una estructura de datos JSON.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.34: Requisito no funcional - Estructura de nivel.

Identificador	**RNF02**
Nombre	Estructura de nivel
Descripción	La estructura es JSON. Se almacenan las variables vidas, tiempo, fondo, autor, siguiente nivel, música y una matriz cuadrada con la codificación numérica del nivel.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.35: Requisito no funcional - Elementos del nivel.

Identificador	**RNF03**
Nombre	Elementos del nivel
Descripción	Cada elemento del nivel está codificado en un número.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.36: Requisito no funcional - Gráficos.

Identificador	**RNF04**
Nombre	Gráficos
Descripción	Los gráficos siguen una temática pixelart en formato de sprites .png.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.37: Requisito no funcional - Fondos de nivel.

Identificador	**RNF05**
Nombre	Fondos de nivel
Descripción	Los fondos de nivel serán imágenes .jpg o .png y se utilizan con la técnica parallax scrolling.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.38: Requisito no funcional - Formato de sonidos.

Identificador	**RNF06**
Nombre	Formato de sonidos
Descripción	Los sonidos y música del juego están en formato .waw o .mp3.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.39: Requisito no funcional - Resolución de pantalla.

Identificador	**RNF07**
Nombre	Resolución de pantalla
Descripción	La resolución de pantalla es adaptable y el usuario puede elegirla en un submenú principal.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.40: Requisito no funcional - Inputs.

Identificador	**RNF08**
Nombre	Inputs
Descripción	Los controles input son adaptables y el usuario puede elegirlos en un submenú principal.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.41: Requisito no funcional - S.O. compatibles.

Identificador	**RNF09**
Nombre	S.O. compatibles
Descripción	Los sistemas operativos son Windows 7/8.1/10, Mac OS X, Linux y Android.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.42: Requisito no funcional - Requisitos de sistema.

Identificador	**RNF10**
Nombre	Requisitos de sistema
Descripción	Los requisitos mínimos de hardware necesarios son: • Procesador 1.5 Ghz. • RAM: 2GB • Almacenamiento: 200 MB
Prioridad	Esencial
Necesidad	Alta

Tabla 4.43: Requisito no funcional - Portabilidad.

Identificador	**RNF11**
Nombre	Portabilidad
Descripción	El juego en cualquiera de sus plataformas es portable y no necesita de instalación. Se distribuye en formato .zip.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.44: Requisito no funcional - Excepciones.

Identificador	**RNF12**
Nombre	Excepciones
Descripción	El juego dispone de tratamiento de excepciones en caso de error que permiten el normal funcionamiento aunque estas sucedan.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.45: Requisito no funcional - Plataforma de desarrollo.

Identificador	**RNF13**
Nombre	Plataforma de desarrollo
Descripción	El proyecto se desarrolla en Unity 5.5.2f1 (64-bit).
Prioridad	Esencial
Necesidad	Alta

Tabla 4.46: Requisito no funcional - Lenguaje de programación.

Identificador	**RNF14**
Nombre	Lenguaje de programación
Descripción	El lenguaje de programación es C#.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.47: Requisito no funcional - Idioma.

Identificador	**RNF15**
Nombre	Idioma
Descripción	El idioma del juego es inglés.
Prioridad	Esencial
Necesidad	Alta

Tabla 4.48: Requisito no funcional - Manual de juego.

Identificador	**RNF16**
Nombre	Manual de juego
Descripción	Se adjunta en la raíz del juego un documento pdf con indicaciones y usabilidad.
Prioridad	Esencial
Necesidad	Alta

4.1.3. Diagramas de flujo

Los diagramas de flujo representan de forma gráfica un algoritmo o proceso. Los símbolos gráficos del flujo del proceso están unidos entre sí con flechas que indican la dirección de flujo del proceso.

4.1.3.1. Diagrama de flujo general

Este diagrama es la representación más global y superficial que abarca el proyecto.

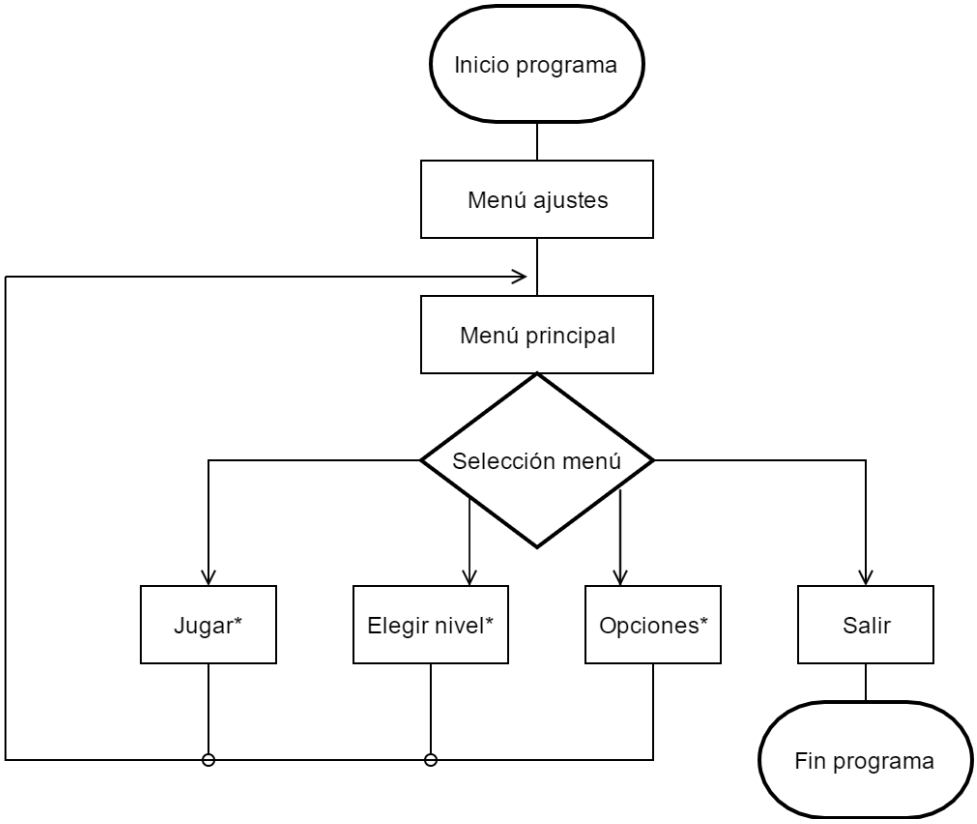

Figura 4.2: *Diagrama de flujo general.*

4.1.3.2. Diagrama de flujo - Jugar

El diagrama de jugar muestra el flujo que el usuario tendrá con la interfaz principal.

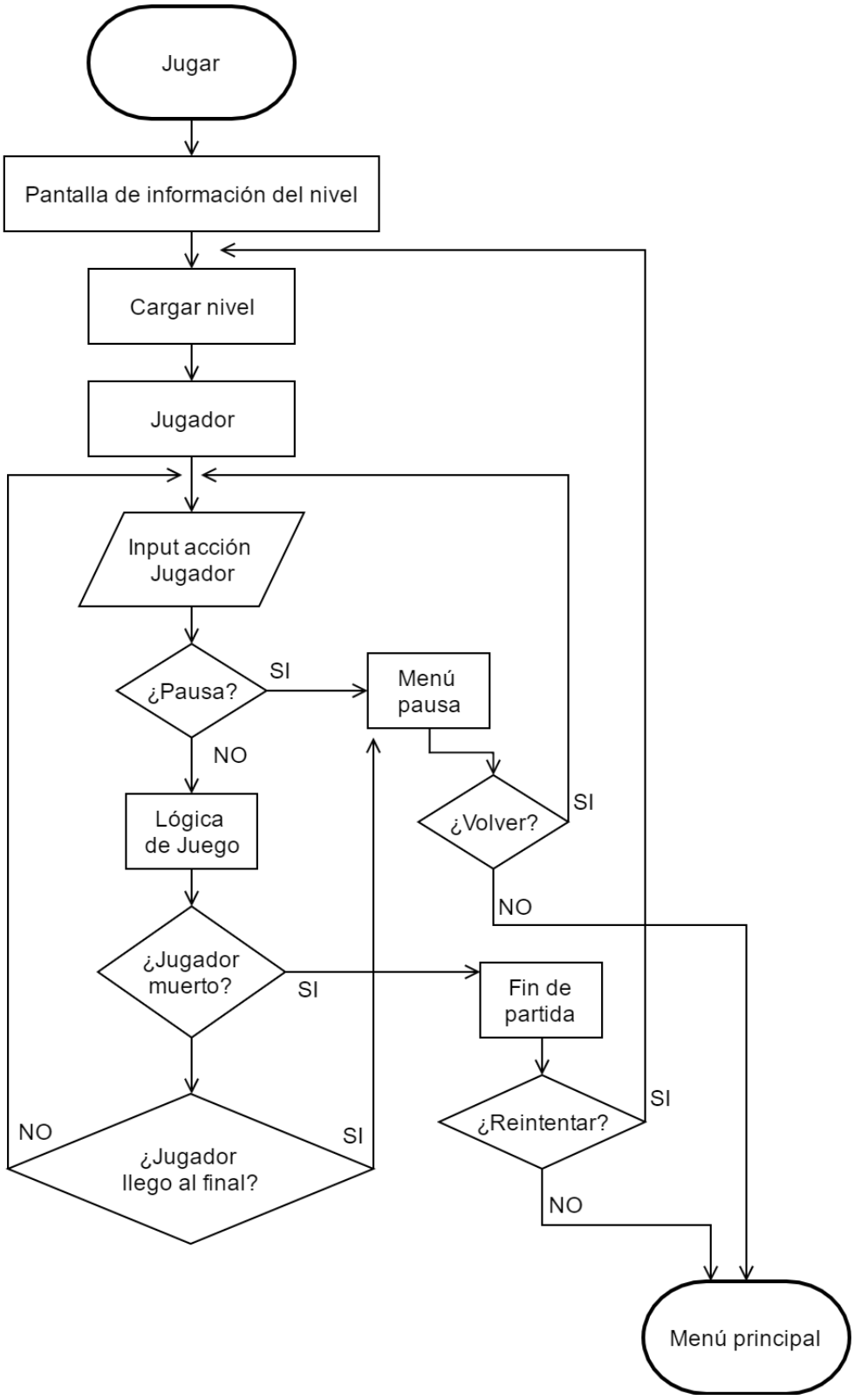

Figura 4.3: *Diagrama de flujo jugar.*

4.1.3.3. Diagrama de flujo - Elegir nivel

Jerarquía de elección de niveles dentro del menú principal.

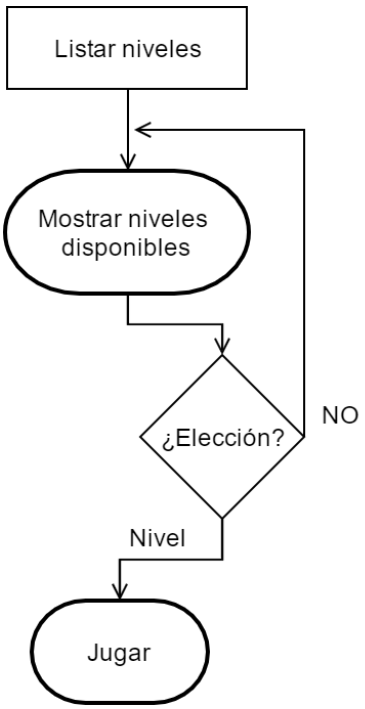

Figura 4.4: *Diagrama de flujo elegir nivel.*

4.1.3.4. Diagrama de flujo - Opciones

Flujo de las opciones disponibles.

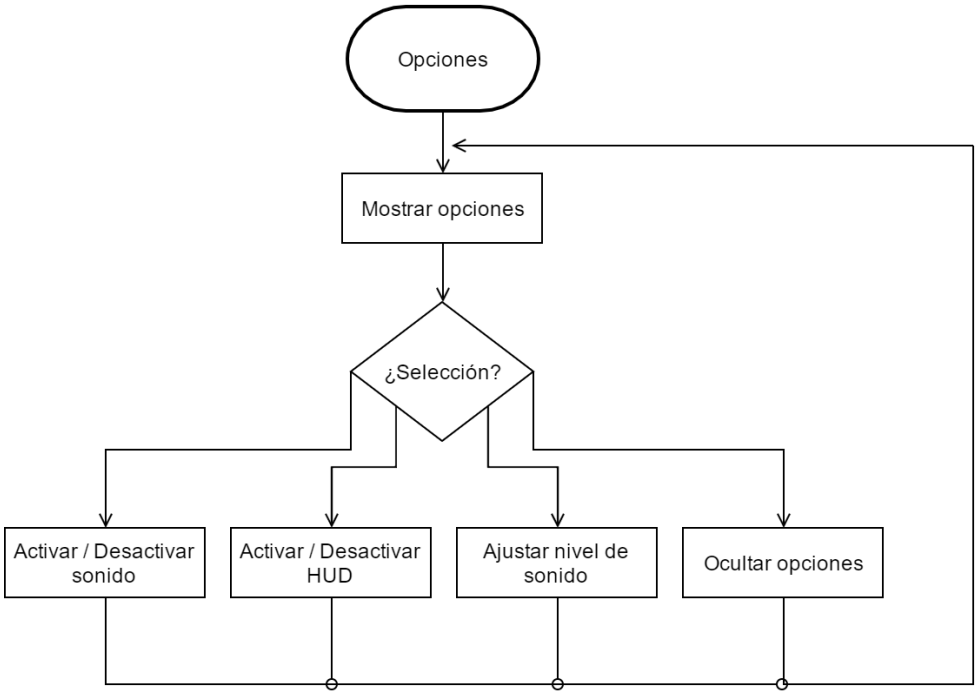

Figura 4.5: *Diagrama de flujo opciones.*

4.1.4. Diagrama de actividad

El diagrama de actividad general de la aplicación muestra la transición y gestión entre las pantallas de la interfaz.

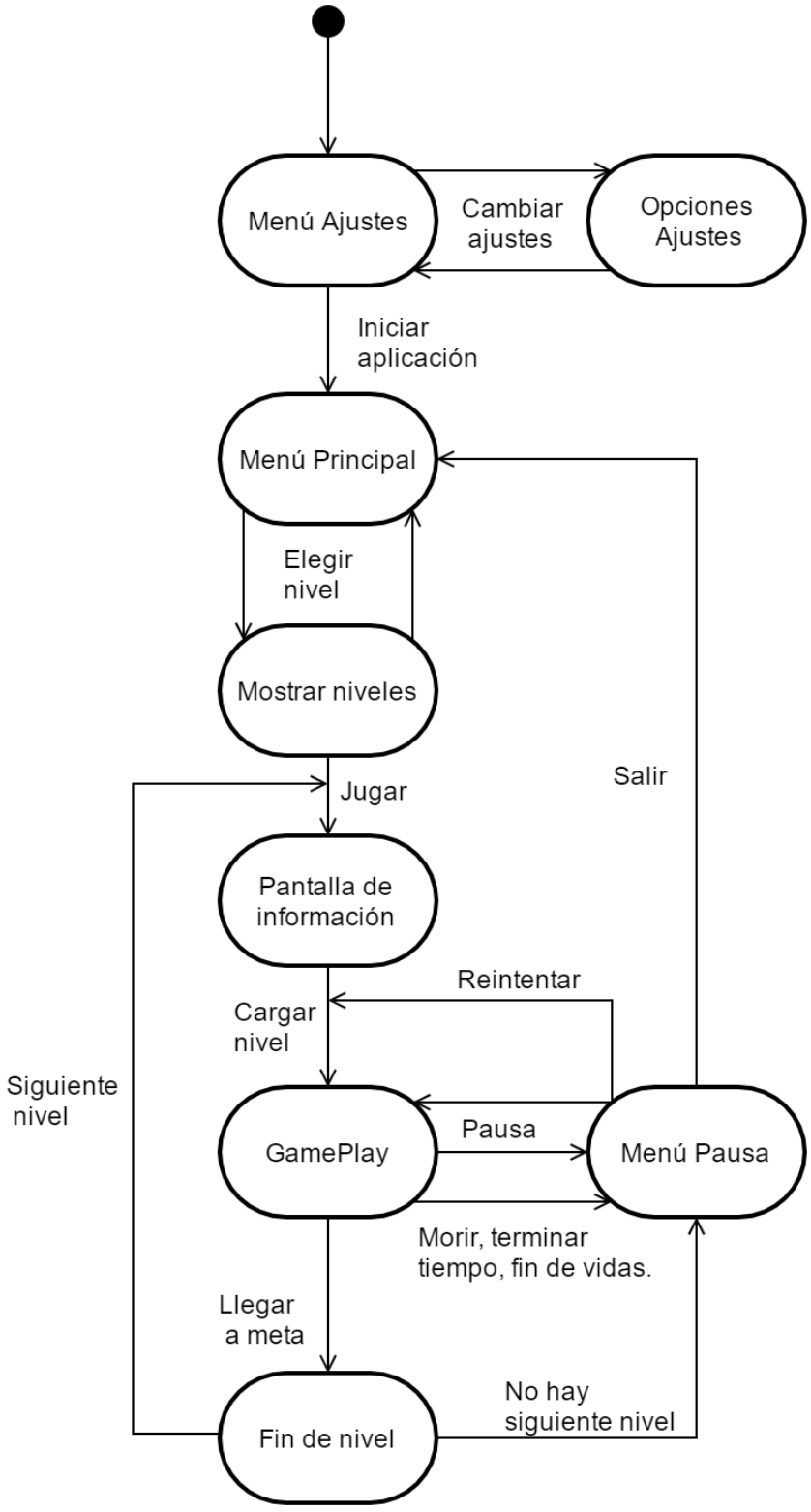

Figura 4.6: *Diagrama de actividad.*

4.2. Diseño conceptual

4.2.1. Arquitectura del sistema

La descripción general de la arquitectura del sistema se muestra a continuación. Cada etiqueta es un módulo y su interacción.

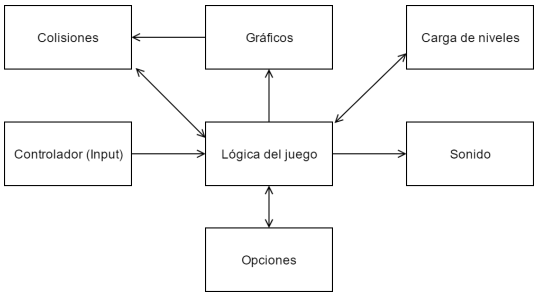

Figura 4.7: *Arquitectura del sistema en modulos*

4.2.2. Diagrama de clases

La disposición y distribución de las clases que se ha seguido corresponde a la codificación del script que permite esa funcionalidad concreta, a continuación se explican los métodos detalladamente.

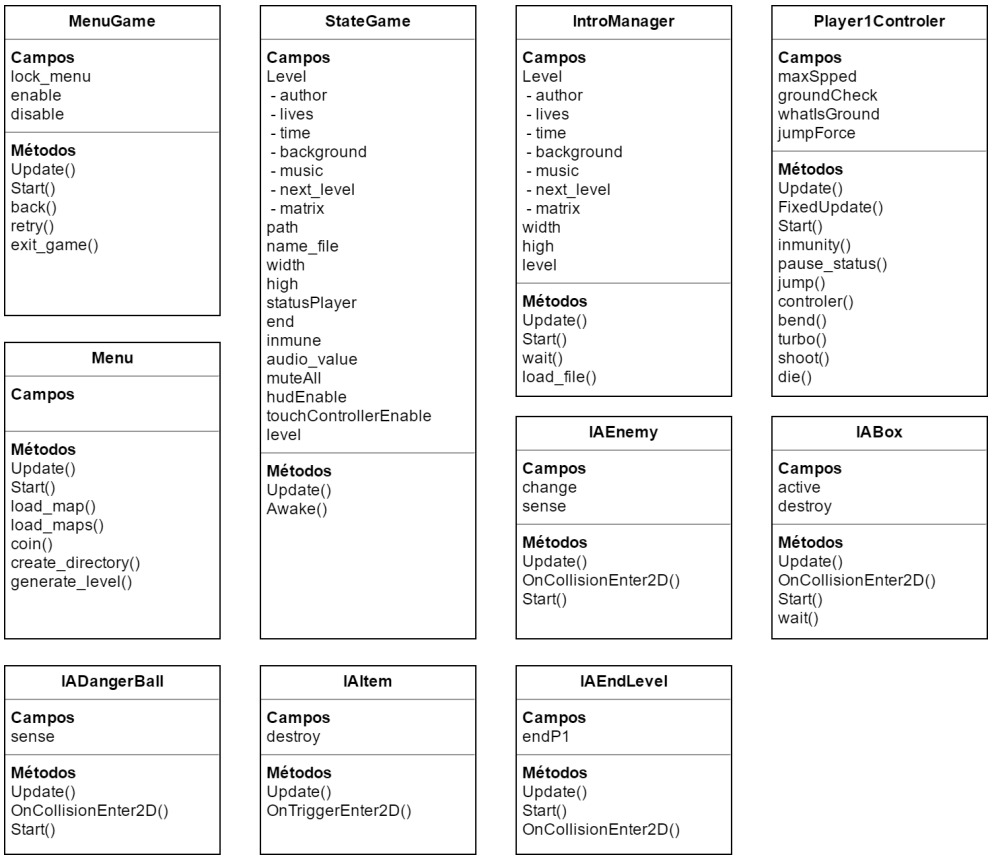

Figura 4.8: *Clases del sistema*

Los métodos de las clases componen la funcionalidad de los elementos del juego. La mayoría de estos contienen el método Start que se identifica como el constructor, y Update que se actualiza a cada tic de reloj y permite comprobar a cada momento cambios e interacciones del elemento.

- **MenuGame**

  Corresponde al menú de pausa del juego una vez se está dentro de un nivel cualquiera.

  - **back:** Volver al juego.
  - **retry:** Reintentar nivel.
  - **exit_game:** Salir del nivel.

- **Menu**

  Menú principal del juego. Permite elegir entre niveles, opciones, etc.

  - **load_map:** Carga el nivel seleccionado y lanza la escena de información del nivel.
  - **load_maps:** Carga en un desplegable el listado de ficheros .map disponibles en la carpeta levels.
  - **create_directory:** Si no hay directorio levels se crea.
  - **generate_level:** Si no hay niveles se generan 3 niveles de ejemplo preprogramados.
  - **coin:** Efecto de sonido de moneda al pulsar en el fondo de la pantalla.

- **StateGame**

  Esta clase se encarga de mantener la información necesaria entre los cambios de escena o niveles (matriz del nivel, vidas, estado del personaje y valores de las opciones modificados)

  - **awake:** Este método permite mantener al elemento siempre en memoria y evitar que se destruya.

- **IntroManager**

  Se encarga de dirigir el funcionamiento de la escena de información cuando se elige un nivel.

  - **wait:** Congela unos segundos la escena para que el usuario pueda leer la información y en segundo plano se pueda cargar el nivel.
  - **load_file:** Función encargada de leer el fichero .map y almacenar la información.

- **Player1Controller**

  Esta clase se encarga de manejar el controlador(input) del jugador. La mayoría de los eventos se ejecutan en el Update.

  - **FixedUpdate:** Permite corregir algunos parámetros de la función Update.
  - **inmunity:** Controla cuando el personaje es inmune a recibir daño.
  - **pause_status:** Controla cuando el personaje debe permanecer en pausa.
  - **jump:** Control de salto.
  - **controller:** Control de movimiento.
  - **bend:** Control de agacharse.
  - **turbo:** Control de correr.
  - **shoot:** Control de disparo.
  - **die:** Detecta cuando el personaje muere.

- **IAEnemy**

  Corresponde a una pequeña inteligencia artificial del enemigo. Siempre avanza y si se choca cambia de sentido.

  - **OnCollisionEnter2D:** Controla las colisiones (es eliminado, se choca con algo o se cae por un precipicio)

- **IABox**

  Clase genérica de las cajas de moneda, powers ups o estrellas.

  - **OnCollisionEnter2D:** Detecta cuando son abiertas y generan el ítem específico.
  - **wait:** Espera unos breves instantes para desactivarse tras ser abierta.

- **IADangerBall**

  Clase genérica de bolas de disparo.

  - **OnCollisionEnter2D:** Detectan la colisión con un elemento y deciden.

- **IAItem**

  Clase genérica de un ítem del juego.

  - **OnTriggerEnter2D:** Cuando el ítem es recogido se dispara un evento.

- **IAEndLevel**

  Se encarga de detectar cuando el jugador ha llegado al final del nivel.

  - **OnCollisionEnter2D:** Cuando el jugador colisiona con el final termina el nivel.

4.3. Implementación

En esta sección se explica el proceso seguido para la programación y funcionamiento de cada una de las partes establecidas del proyecto, además de las ideas iniciales, evolución de estas y decisiones de diseño hasta alcanzar el resultado final.

4.3.1. Ficheros JSON y carga de niveles

La idea principal del proyecto era generar una carga básica de un fichero de texto y que el programa fuera capaz de adaptar esos datos leídos y construirlo en el juego. Los textos planos se podían configurar como una matriz cuadrada donde cada elemento (caracteres) puede ser acotado en el código. Las versiones iniciales tomaban dos caracteres alfanuméricos, permitiendo un gran número de combinaciones. El código se centraba en leer el contenido línea a línea e ir cortando y almacenando cada dos caracteres en una matriz para posteriormente hacer una comparación de cada código y construirlos.

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000B3000000B
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
000000000000M1000000C0C0C0C000000000000001
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
```

Figura 4.9: *Ejemplo nivel txt.*

Los principales errores de diseño que tenía este método, era la incapacidad de meter más información en el propio archivo de texto, dejando que opciones como la vida y el tiempo no fueran configurados por el creador del nivel sino por un jugador desde el menú de opciones.

Las alternativas eran cambiar el tipo de fichero, las opciones que se plantearon fueron:

- Archivos binarios, pero estos no se podían editar con un editor de texto y perdían esa propiedad característica.
- Introducir líneas adicionales en el archivo de texto, pero resultaba algo tosco y difícil de configurar parámetros vacíos o que no se querían incluir en el nivel.
- Utilizar un formato de intercambio de información como XML, YAML o JSON dado que permiten ser editados fácilmente por cualquier editor y usan el concepto de variable para almacenar información.

La decisión final fue JSON (JavaScript Object Notation) por un conocimiento previo en Python y la capacidad de estructurar matrices cuadradas que son perfectas para representar un mapeado 2d en scroll lateral. Este tipo de estructuración permite introducir nuevos campos para almacenar el número de vidas, el tiempo disponible para pasar el nivel, el autor del nivel que se muestra antes de iniciarlo, la música de fondo, la imagen de fondo y un posible siguiente nivel que será cargado al finalizar el actual con éxito.

```
{
"author":"dani",
"lives":05,
"time":400,
"background":01,
"music":03,
"next":"single [1-2]",
"matrix":
[
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,04,00,00,00,01,03,01,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,12,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,70,00,00,00,00,00,00,00,00,00,00,00,00,00,76,00,00],
        [09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09],
        [08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08],
        [08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08]
]
}
```

Figura 4.10: *Ejemplo nivel con estructura JSON.*

Para la implementación se usó una librería gratuita disponible dentro de los assets de Unity que se ofrece a su comunidad. Algunos de los parámetros para la lectura de la matriz tuvieron que ser modificados, la librería está adaptada para el uso de vectores unidimensionales y tiene algunas dificultades a la hora de leer dimensiones superiores. La librería se llama JSONObject y su enlace es el siguiente. `https://www.assetstore.unity3d.com/en/#!/content/710`
La función de lectura de los ficheros JSON se muestra a continuación.

```
void load_file(String path)
{
    try
    {

        String file = File.ReadAllText(path);
        JSONObject json = new JSONObject(file);

        // Lectura de variables
        level.author = json[0].str;
        level.lives = (int)json[1].n;
        level.time = (int)json[2].n;
        level.background = (int)json[3].n;
        level.music = (int)json[4].n;
        level.next_level = json[5].str;
        int i = 0, j = 0;

        // Lectura de la matriz
        while (json[6][i] != null)
        {
            while (json[6][i][j] != null)
            {
                j++;
            }
            width = j;
            j = 0;
            i++;
        }
        high = i;
        level.matrix = new int[high, width];
        for (i = 0; i < high; i++)
        {
            for (j = 0; j < width; j++)
            {
                level.matrix[i, j] = (int)json[6][i][j].n;
            }
        }

    }
    catch (Exception e)
    {
        Debug.Log(e);
    }
}
```

Figura 4.11: *Lectura JSON.*

4.3.2. Instanciación de elementos

Como se comentó en la sección anterior, una vez leída la estructura del fichero JSON y almacenada la matriz codificada de los elementos que componen el mapa, el proceso necesario para generarlo dentro del juego es mediante una instanciación asociada a un prefab diseñado anteriormente para ese elemento. Los códigos son numéricos sin límite establecido, propiedad conseguida gracias a los ficheros JSON, lo que establece poder integrar al juego infinitos elementos para la creación de niveles.

La implementación del código está basada en un switch que identifica la id del elemento y localiza la ruta del prefab. Este prefab es instanciado en la coordenada correspondiente a la matriz empezando en el punto (0 , 0) y tomando desplazamientos según el tamaño del elemento instanciado.

```
// Nuevo elemento
mapped[i, j] = new Cell();
// Identificar ID elemento
GameObject newcell = type_block(level.matrix[i, j], position);
// Instanciar elemento en el nivel
mapped[i, j].block = Instantiate(newcell,
    new Vector3(position.x, position.y, newcell.GetComponent<Transform>().position.z),
    Quaternion.Euler(0, 0, 0)) as GameObject;
```

Figura 4.12: *Instanciar elemento.*

4.3.3. Codificación y composición de componentes

Todos los componentes instanciados se diseñan y construyen en un tipo de objeto denominado prefab. Estos en su mayoría se componen de 4 subcomponentes característicos localizados en el Inspector.

- **Transform:** Contiene la información de posicionamiento del objeto (Posición, Rotación y Escala).
- **Sprite renderer:** Contiene la imagen asociada al elemento y algunos parámetros configurables como el material, la orientación y el modo de dibujado.

- **Collider 2D:** Permite asociar al elemento un tipo de colisión, por lo general son cuadradas o circulares y si se incluye un componente rigidbody se consigue la propiedad de físicas y acceso al controlador de movimiento.

- **Script:** Fichero de código que incluye las instrucciones específicas que tendrá el prefab. Está compuesto siempre por dos funciones principales. Start, que sería una especie de constructor, se lanza en la creación del objeto y configura todos los parámetros necesarios para su funcionamiento, y Update, función que se lanza en cada tic de reloj y permite introducir mecánicas de continuidad.

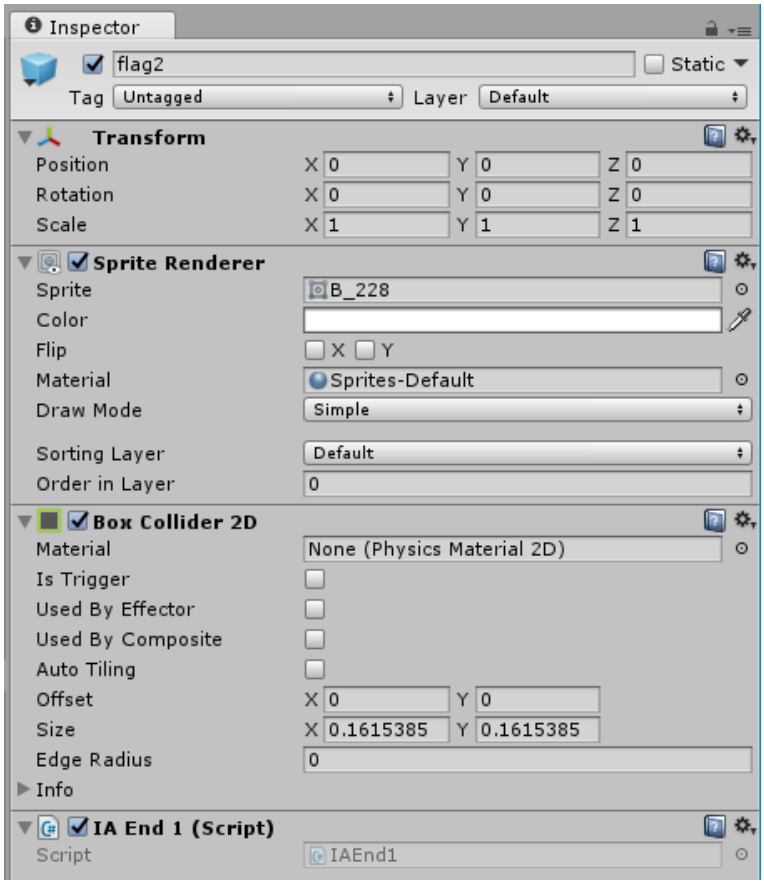

Figura 4.13: *Ejemplo Inspector de un prefab.*

Los prefabs están clasificados en 5 secciones para poder localizarlos de forma cómoda.

- **Building:** Todo tipo de construcciones, desde tipos de suelo, bloques, cajas, banderas, monedas, etc.
- **Character:** Personajes disponibles en sus distintas fases de evolución.
- **Danger:** Elementos que dañan o son lanzados por el jugador.
- **Enemies:** Todos los enemigos disponibles.
- **Powers:** Elementos que permiten evolucionar al personaje, powers up, estrellas de inmunidad y flores de fuego.

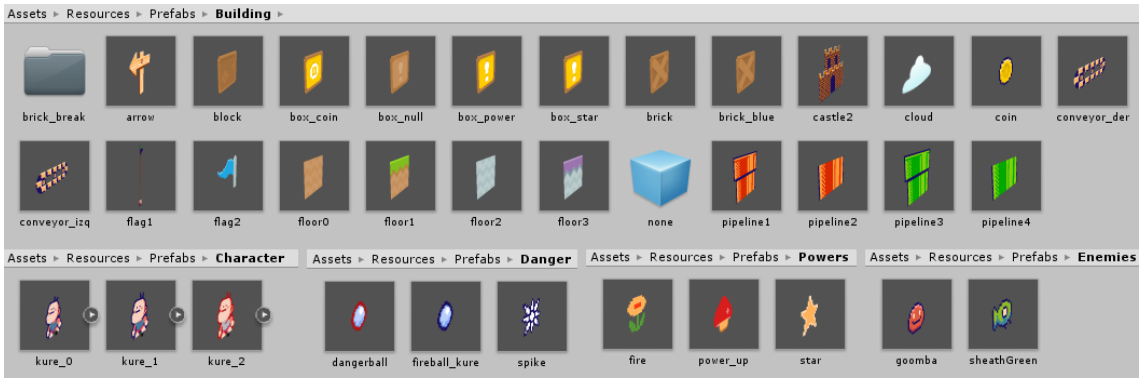

Figura 4.14: *Prefabs disponibles agrupados por secciones.*

4.3.4. Controlador de personaje

El input (Controlador de personaje) se divide en movimiento horizontal, correr, saltar, disparar y agacharse. Todo este proceso está programado en el Script agregado al prefab del personaje el cual está dotado de un circulo de colisiones (Circle Collider) que detecta cuando tiene algo bajo sus pies, como un bloque para posarse en el suelo, un enemigo para aplastarlo o un bloque que golpear. Cada acción del controlador que se realiza tiene asociada una animación de sprites y un determinado sonido. A continuación se definen las funciones especificas del controlador y un fragmento de su implementación.

- **Movimiento horizontal:** es manejado por el componente Rigidbody 2D que se encuentra dentro del prefab del personaje. Este componente capta cuando es pulsada una tecla de movimiento y añade una fuerza horizontal en ese sentido para desplazar al personaje. Las teclas designadas son WASD y las teclas de dirección, además en dispositivos Android aparece un joystick que adapta este control.

```
GetComponent<Rigidbody2D>().velocity = new Vector2(move * maxSpeed, GetComponent<Rigidbody2D>().velocity.y);
```

Figura 4.15: *Código de movimiento horizontal*

- **Correr:** Exactamente igual que el movimiento horizontal pero el desplazamiento es mayor cuando se pulsa la tecla F o el botón B en Android.

```
public void turbo()
{
        if (maxSpeed >= 1.25f)
        {
            jumpForce = 180f;
            anim.SetBool("Turbo", true);
        }else
        {
            turboMax += 0.0005f;
            maxSpeed += turboMax;
        }
}
```

Figura 4.16: *Función de correr*

- **Saltar:** Cuando se oprime la tecla de salto se ejerce una fuerza vertical hacia arriba de desplazamiento. El salto solo se puede ejecutar si el personaje ha tocado el suelo anteriormente. Las teclas asignadas son el botón espacio y el botón A en Android.

```
public void jump()
{
    if (grounded && !anim.GetBool("Bend") && !pause)
    {
        anim.SetBool("Ground", false);
        GetComponent<Rigidbody2D>().AddForce(new Vector2(0, jumpForce));
        if (anim.GetBool("Turbo"))
        {
            audioSource.clip = Resources.Load("AudioEffects/spring_jump") as AudioClip;
            audioSource.Play();
        }
        else
        {
            audioSource.clip = Resources.Load("AudioEffects/jump") as AudioClip;
            audioSource.Play();
        }
    }
}
```

Figura 4.17: *Función de salto*

- **Disparar:** La opción de disparo solo está disponible tras tocar una flor de fuego. Al oprimir la tecla de disparo se genera una bola con una determinada fuerza que sale en frente del personaje. Las teclas asignadas son F y B en Android.

```
public void shoot()
{
    if ((GameObject.Find("StateGame").GetComponent<StateGame>()).statusPlayer1 == 2 && !anim.GetBool("Bend") && !pause)
    {
        int sense = 1;
        // Comprobar sentido de kure
        if (gameObject.GetComponent<SpriteRenderer>().flipX)
        {
            sense = -1;
        }
        Vector3 pos = new Vector3(gameObject.transform.position.x+ (sense * 0.1f),
            gameObject.transform.position.y, gameObject.transform.position.z);

            if(GameObject.Find(gameObject.name+"/FireBall").transform.childCount < 3)
            {
                anim.SetBool("Shoot", true);
                Instantiate(Resources.Load("Prefabs/Danger/fireball_kure"), pos, Quaternion.Euler(0, 0, 0));
            }
    }
}
```

Figura 4.18: *Función de disparo*

- **Agacharse:** Cuando se oprime la tecla de agacharse el personaje se inmoviliza y pasa a ocupar un bloque de alto sin depender del estado en el que se encuentre, la única condición es que esté posado en un bloque.

```
void bend()
{
    down = Input.GetAxis("Vertical");
    float downTouch = CrossPlatformInputManager.GetAxisRaw("Vertical");
    if ((down < 0  || (downTouch < -0.9 && Math.Abs(CrossPlatformInputManager.GetAxisRaw("Horizontal"))>=0.2))&& anim.GetBool("Ground"))
    {
        // Para evitar efecto hielo
        anim.SetBool("Bend", true);
        if ((GameObject.Find("StateGame").GetComponent<StateGame>()).statusPlayer1 >= 1)
        { gameObject.GetComponents<CircleCollider2D>()[1].enabled = false;}
    }
    else
    {
        anim.SetBool("Bend", false);
        if ((gameObject.name.Equals("kure_1(Clone)") || gameObject.name.Equals("kure_2(Clone)"))
            &&(GameObject.Find("StateGame").GetComponent<StateGame>()).statusPlayer1 >= 1)
        { gameObject.GetComponents<CircleCollider2D>()[1].enabled = true;}
    }
}
```

Figura 4.19: *Función de agacharse*

4.3.5. Sprites, Audios e Interfaz

Un Sprite es una imagen o conjunto de imágenes que permite representar visualmente elementos del juego. Las propiedades características del Sprite es que son esencialmente para juegos en 2 dimensiones y entre el conjunto usado tienen una resolución parecida que permite integrarse unos con otros. El banco de sprites que se ha utilizado en el proyecto es «`https://opengameart.org`», que dispone de material open source creado por su comunidad, en concreto el elegido sigue una temática pixelart (se destaca por el uso de resoluciones muy bajas, pocos colores y animaciones con pocos frames). Las fuentes especificas son:

- **Bloques, suelo y cajas:** `https://opengameart.org/content/platformer-art-pixel-redux`
- **Enemigos:** `https://opengameart.org/content/platformer-baddies`
- **Personaje:** `https://opengameart.org/content/superpowers-assets-characters`
- **Texturas propias:** Todos los sprites han sido retocados y modificados previamente con la herramienta Adobe Photoshop. Algunos de los elementos usados son de creación propia.

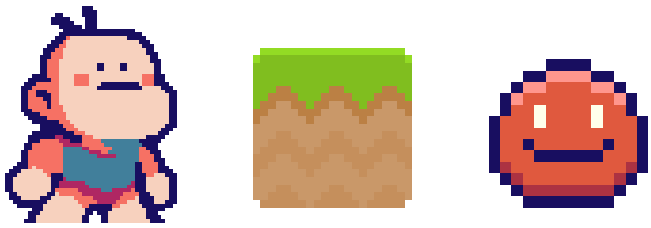

Figura 4.20: *Ejemplo de Sprites finales. (Personaje/Suelo/Enemigo)*

Los efectos de audio usados en el proyecto (Salto, Disparo, etc) están sustraídos de (`https://opengameart.org/content/512-sound-effects-8-bit-style`) que dispone de una amplia variedad de sonidos a 8 bits de libre uso.

La música de fondo es creación del autor Mudeth (`https://mudeth.bandcamp.com/album/antibirth-ost`), se le pidió permiso de usar las composiciones en el proyecto. La música forma parte de la OST del mod Antibirth del juego The Binding of Isaac - Afterbirth.

La interfaz se ha diseñado con el propio motor Unity y una librería de Material UI (`https://www.assetstore.unity3d.com/en/#!/content/51870`) para dar un efecto material design agradable para el usuario (limpia, ordenada y sencilla).

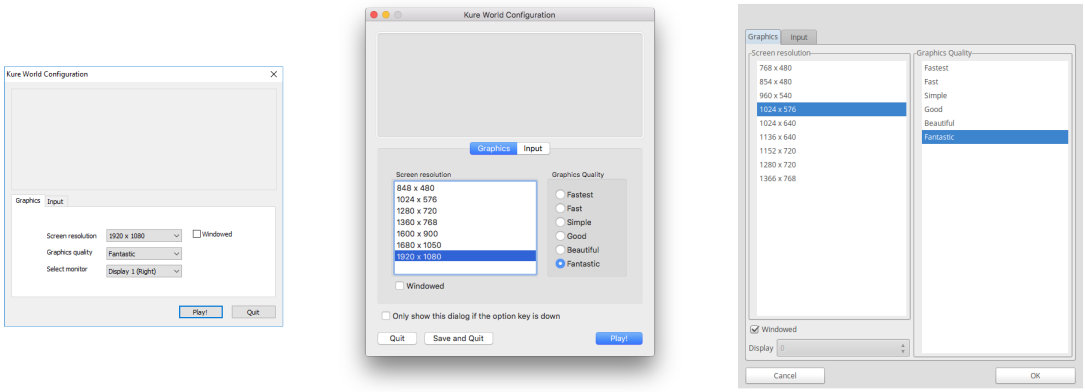

Figura 4.21: *Interfaz ajustes previos (Windows, Mac OS X, Linux)*

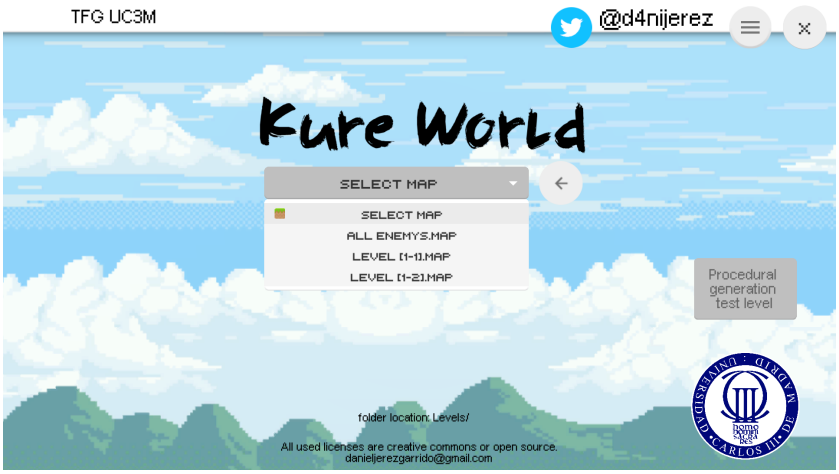

Figura 4.22: *Interfaz menú principal*

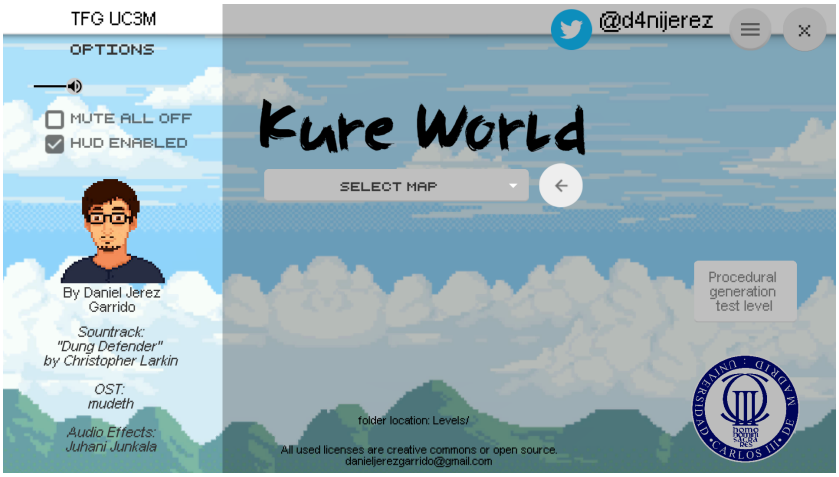

Figura 4.23: *Interfaz opciones*

Figura 4.24: *Interfaz información previa del nivel*

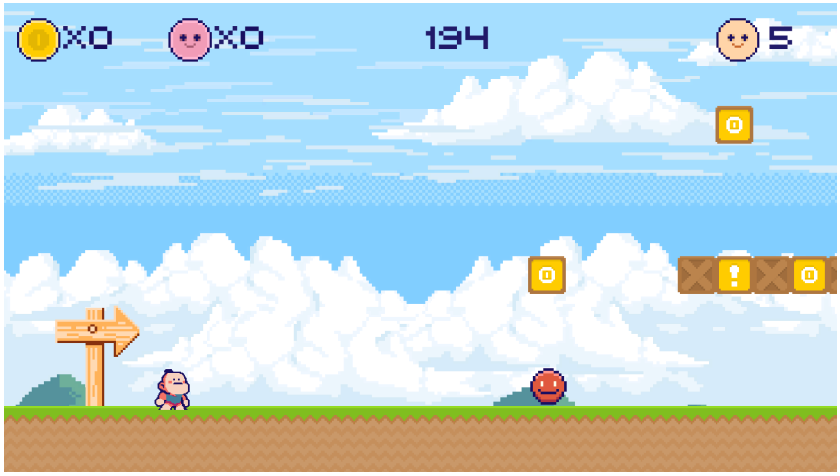

Figura 4.25: *Interfaz del juego en un nivel*

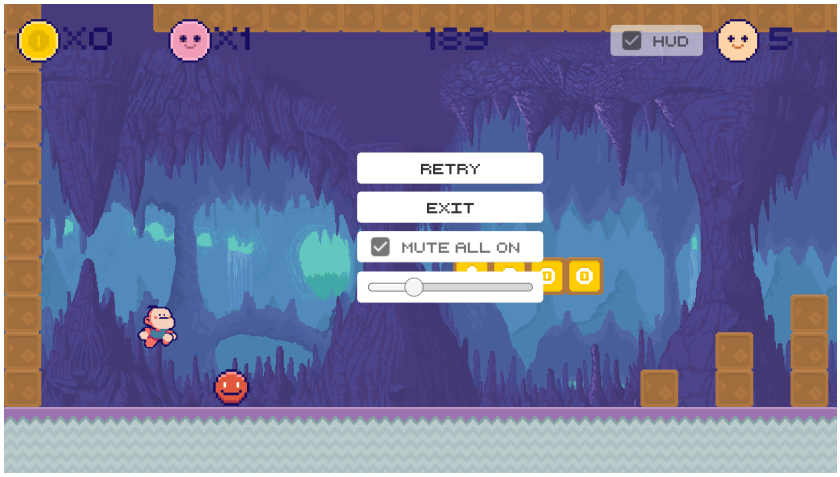

Figura 4.26: *Interfaz menú de pausa*

4.3.6. Autómata de animación y esqueleto de elementos

Los prefabs que requieren una serie de animaciones en situaciones concretas necesitan de un autómata que gestione el estado actual y la transición a un estado nuevo. En concreto el autómata del personaje es el más elaborado, su estado inicial por defecto es stop y cuando su velocidad es superior a 0.01 transita a run (correr) o jump (saltar) si se ha presionado la tecla de salto y el eje Y ha sido modificado. Desde run se puede correr (turbo) y correr saltando (turbo jump). Desde salto solo se puede caer (fall) o pararse nuevamente (stop). Algunas decisiones se pueden tomar desde cualquier estado (any state) como agacharse (bend), disparar (shoot) o ganar en el nivel (win).

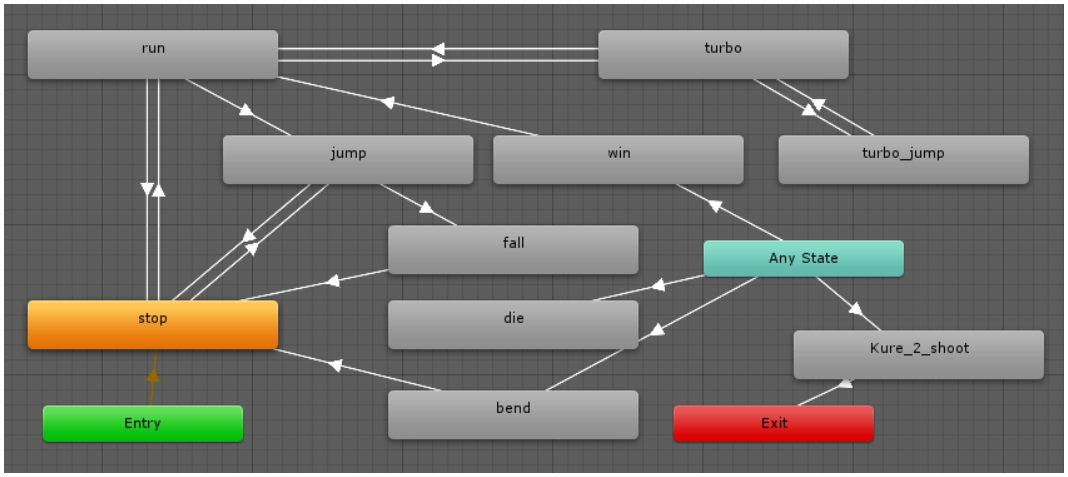

Figura 4.27: *Autómata de movimiento del personaje*

Los objetos además incluyen un esqueleto que les permite detectar estas situaciones. El personaje tiene dos collider, el más grande es el cuerpo, que detecta colisiones con cajas y enemigos y el más pequeño, que son los pies, sirve para detectar cuándo está tocando suelo. La IA (Inteligencia Artificial) de los enemigos es simple, solo avanzan si el personaje está a una determinada distancia, se chocan con algo cambian de sentido (esto se detecta con los colliders circulares colocados en los laterales), para saber si son aplastados disponen de un collider cuadrado en la parte superior del cuerpo. La caja también dispone de dos collides cuadrados, uno para saber cuándo está el personaje posado encima y otro cuando la pulsa desde abajo. El resto de elementos también tienen esqueleto similar a los explicados.

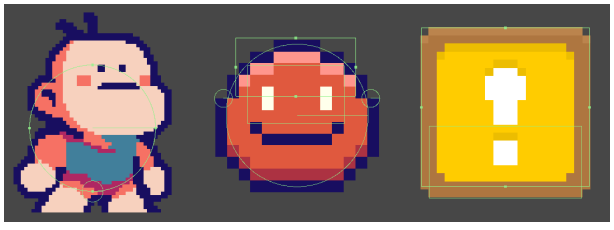

Figura 4.28: *Esqueleto(Colliders) que componen al personaje, enemigo y caja*

4.3.7. Estudio y pruebas de generación procedural

Para las pruebas e implementación de un algoritmo procedural se buscó tipos de juegos similares al objetivo del proyecto. Desde un punto de vista actual, no hay gran variedad de estos que faciliten su código o algoritmo, y el resultado final de calidad no es el esperado debido a la complejidad que conlleva combinar tantos elementos a decisiones casi aleatorias [10].

Las tres propiedades que un algoritmo procedural debe cumplir para la generación de un nivel son:

- **Viabilidad:** Un nivel tiene que tener principio y fin.
- **Diseño:** Deber ser un diseño interesante y bien construido.
- **Dificultad:** Adaptable al jugador en concreto que lo está jugando.

Cada una por separado es fácil de cumplir, pero la combinación de las tres supone una gran dificultad de perfeccionamiento. Los juegos de referencia que se adaptan a estas ideas dentro del género plataformas 2D son pocos y cabe destacar que todos son indies.

- **Spelunky:** Creado por Derek [5] en 2008, combina niveles que son cuevas con generación procedural de enemigos, escenarios, tiendas, etc. La característica del algoritmo que se usó es su sistema de matrices cuadradas que componen el escenario, satisfaciendo siempre un camino posible a su salida. Es el mayor referente de este género y el primer punto de partida. A continuación se definen algunos.

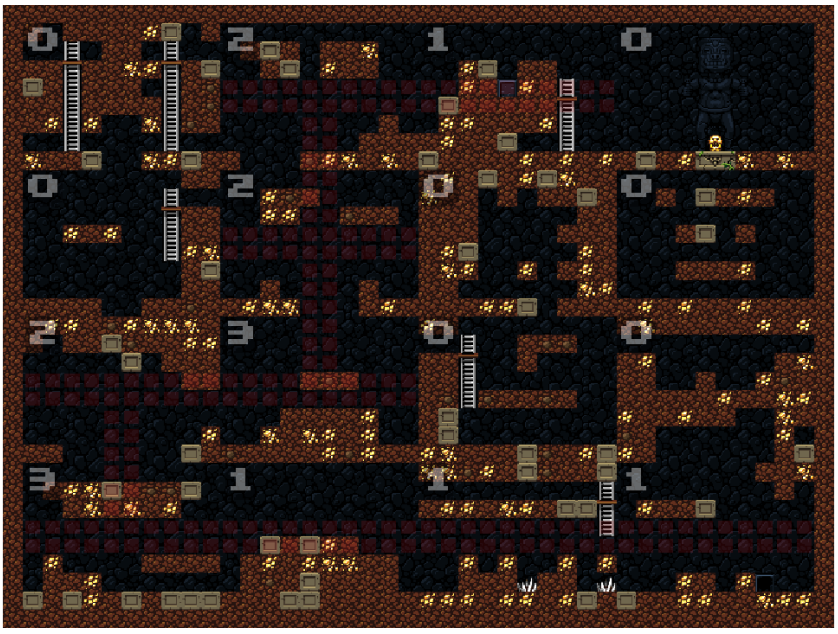

Figura 4.29: *Imagen de generación de nivel con el algoritmo de Spelunky.*
fFuente: `http://tinysubversions.com/spelunkyGen/`

- **Cloudberry Kingdom:** Desarrollado en 2013 por Jordan Fisher y destacado por generar un algoritmo procedural adaptativo según el nivel de habilidad del jugador donde se incluían habilidades en el personaje y alteración de las físicas del juego [10].

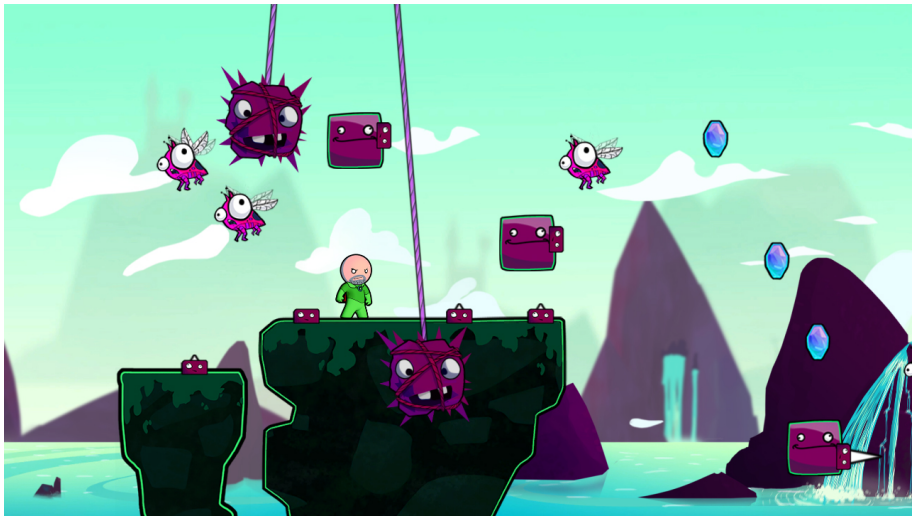

Figura 4.30: *Imagen del juego Cloudberry Kingdom*

- **Downwell:** Desarrollado por Moppin y producido por Devolver Digital. El juego basa su idea en un personaje que se mueve en scroll vertical y tiene que ir bajando por un pozo que se genera de forma procedural.

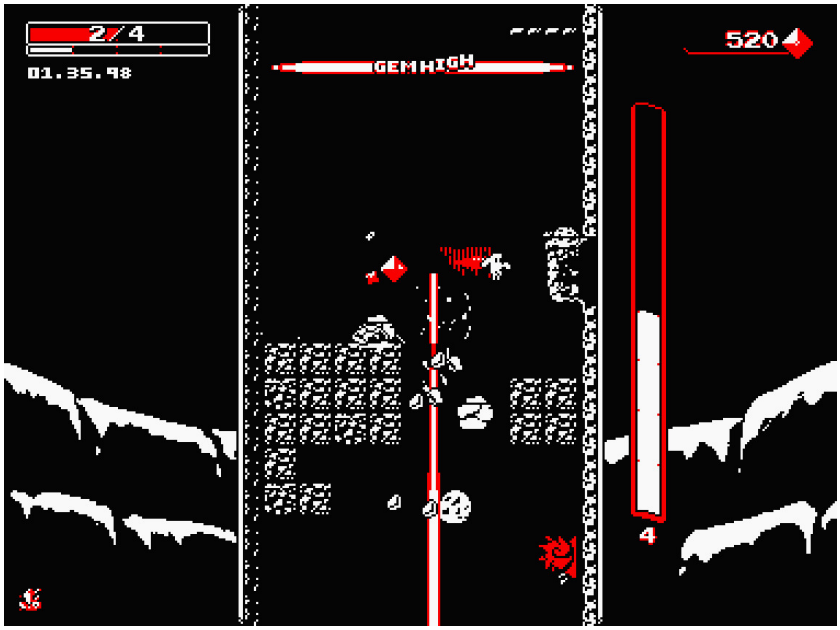

Figura 4.31: *Imagen del juego Downwell*

Tomando estas ideas, el punto de partida fue de búsqueda y experimentación con el algoritmo de Spelunky, pues de los juegos mencionados es el único que tiene el código liberado. Las principales dificultades son que el juego está construido con el motor GameMaker y sin licencia del programa no es posible visualizarlo, además de que, el lenguaje de programación es distinto y su traducción es costosa y compleja. La segunda opción es probar adaptaciones de este algoritmo que usuarios han realizado en Unity, pero los resultados son poco fieles al original y se basan en construir terrenos aleatorios acotados por una malla de matrices que no proporcionan ningún reto al jugador pues los obstáculos son mínimos y hay carencia de otros elementos como enemigos.

Figura 4.32: *Nivel generado con autómata celular.*
Fuente: `https://unity3d.com/es/learn/tutorials/projects/procedural-cave-generation-tutorial/cellular-automata?playlist=17153`

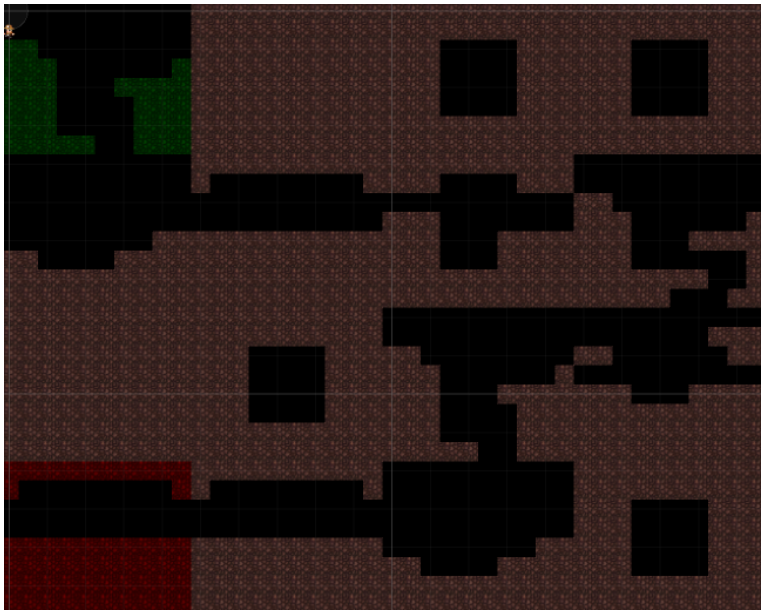

Figura 4.33: *Nivel generado con el algoritmo de Spelunky adaptado en unity (Matriz verde inicio, matriz roja final).*
Fuente: `https://github.com/gholaday/Procedural-Level-Generation`

Las pruebas posteriores fueron de algoritmos inventados o creados por usuarios. Por lo general ninguno era especialmente bueno, pues son métodos realmente complejos y sufren demasiadas carencias de continuidad además de que no se asemejaban a la estética del juego. Se optó por la modificación de un algoritmo (Fuente: `https://github.com/stuartlong/chocolatebox`) que divide el mapeado en secciones que incluyen obstáculos como precipicios o elevaciones que dependía de una semilla afectada por un parámetro de dificultad creciente según se iban superando los niveles.

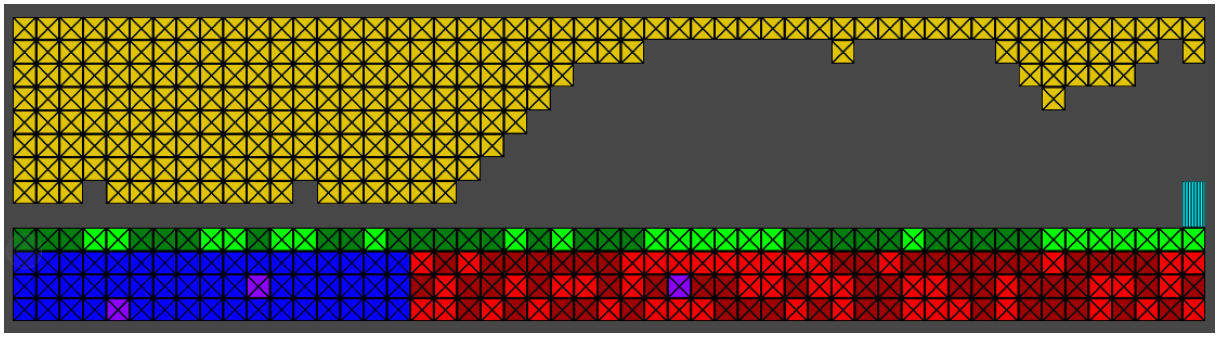

Figura 4.34: *Nivel procedural (1 sección y dificultad baja)*

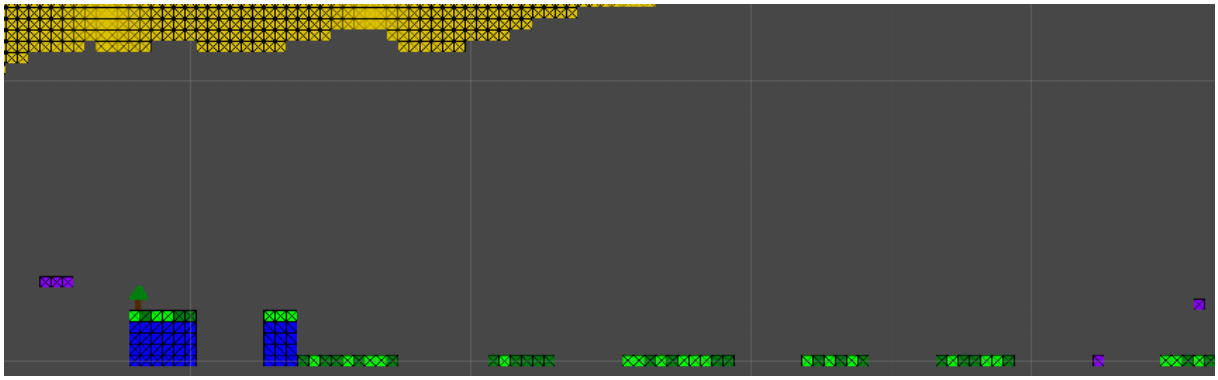

Figura 4.35: *Fragmento de nivel procedural (3 secciones y dificultad alta)*

Aunque los resultados eran mejores de los esperados, la codificación estaba en una versión de Unity inferior a la del proyecto, el código tenía gran cantidad de fallos que lo hacían inestable y la integración al juego resultó imposible pues los prefabs no llegaban a ser compatibles, las físicas estaban planteadas de otra manera y la dimensionalidad era superior, lo que mostraba un escenario inmenso y un personaje diminuto. El algoritmo elegido para implementar en el proyecto fue el modificado de Spelunky, explicado detalladamente a continuación.

4.3.8. Generación procedural implementada

El algoritmo de generación procedural implementado se basa en el proyecto modificado (`https://github.com/gholaday/Procedural-Level-Generation`).

El algoritmo realiza un escenario formado por matrices predefinidas. Cada matriz esta formado por dos elementos (0, 1), el 0 es para los huecos vacíos y el 1 para los elementos. A su vez estas matrices se clasifican en 0, 1, 2 y 3.

- **0:** Matriz sin huecos o con huecos interiores pero sin caminos.
- **1:** Matriz con caminos horizontales.
- **2:** Matriz con caminos verticales.
- **3:** Matriz con caminos superiores y horizontales.

Figura 4.36: *Ejemplo de matrices predefinidas por secciones.*

El conjunto de las matrices forman a su vez una matriz superior cuyas especificaciones se fijaron de 5x5 a 9x9 rangos variables y elegidos aleatoriamente que definen el tamaño del nivel. Este conjunto de matrices genera un camino que dispone una matriz como el inicio y una matriz final, señalada en rojo, como la meta del nivel.

La elección de matrices es aleatoria, se elige una posición de inicio y se coloca una matriz 2 o 3. Las matrices 1 solo pueden ir en laterales, las matrices 2 en posiciones inferiores y las matrices 3 en ambas, de esta forma siempre se genera un camino posible variable en longitud. El resto de posiciones libres se completan con matrices 0 que sirve para rellenar los huecos de la matriz.

Cuando la matriz superior está definida se itera sobre ella y se va instanciando cada elemento. Si el elemento es un 1 se carga un prefab de terreno (bloque, caja, suelo, etc), si es un cero (hueco libre) se instancia un prefab vació, enemigo, caja, moneda, etc. Los porcentajes de estas instanciaciones son parámetros variables y aleatorios e incrementan la dificultad y el resultado final del nivel.

Dado que este tipo de generación de niveles es experimental, se incluyó un botón de acceso en el menú principal como prueba de ensayo y test. Una vez dentro del menú se eligen de forma aleatoria los parámetros vidas, música, tiempo, fondo y dimensiones del nivel para su posterior generación pulsando el botón "Go!".

Figura 4.37: *Menú opcional de las pruebas de generación de niveles procedurales.*

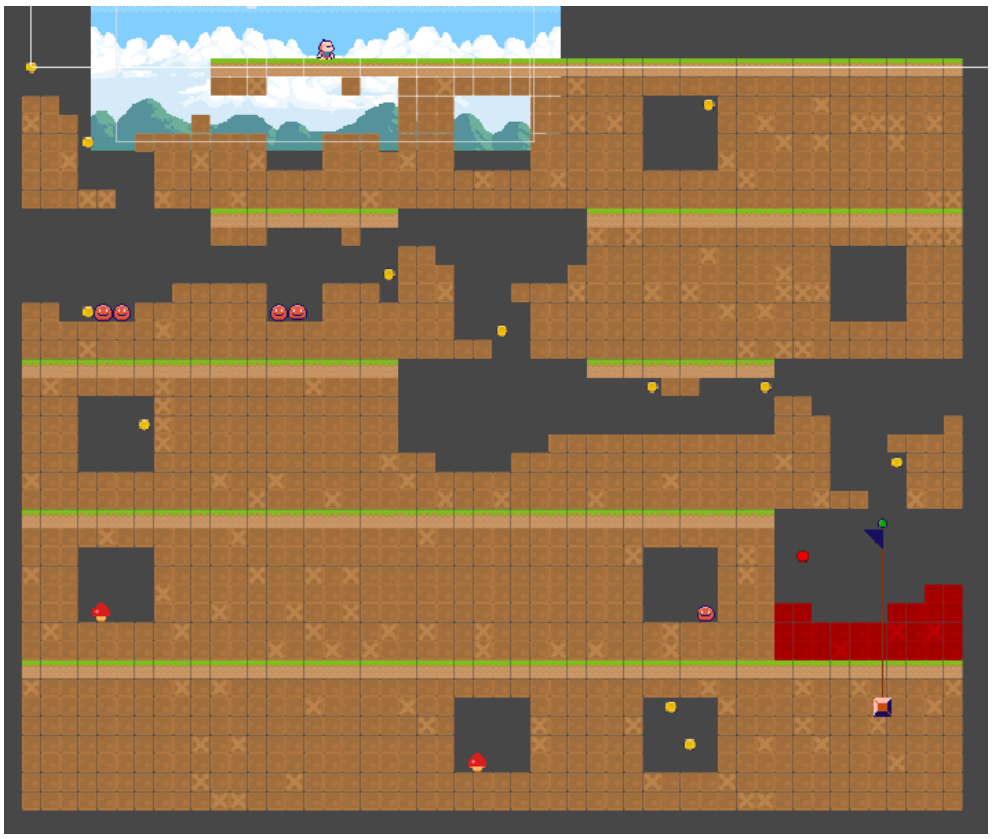

Figura 4.38: *Ejemplo 1 : Nivel generado proceduralmente (5x5).*

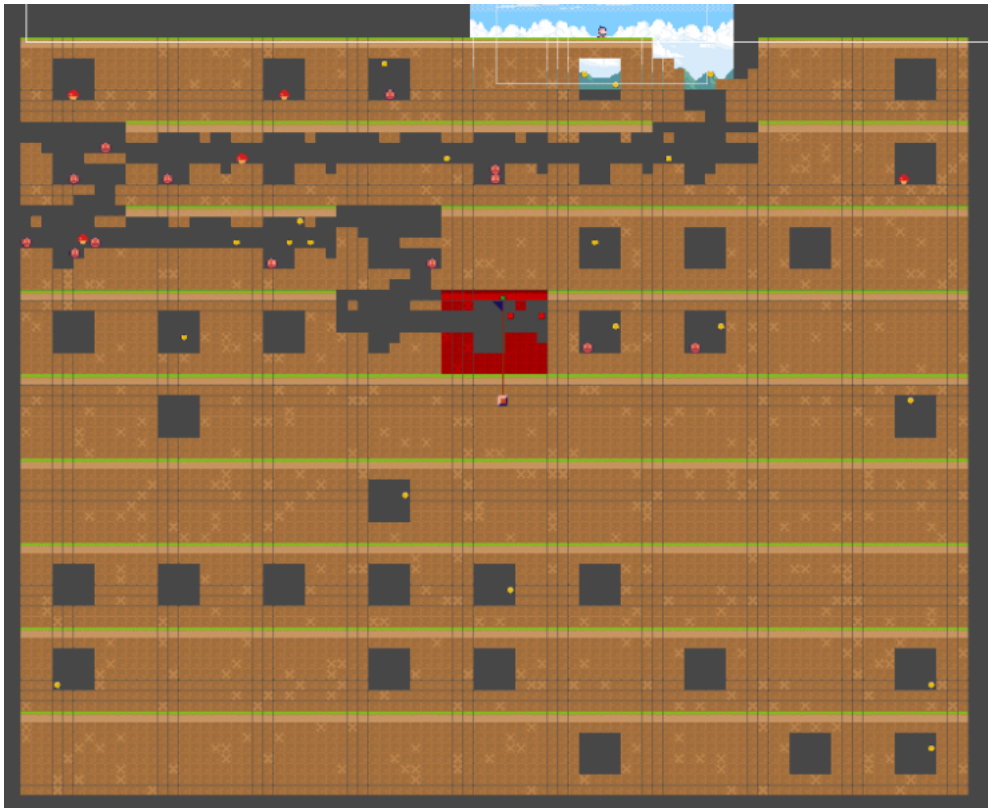

Figura 4.39: *Ejemplo 2 : Nivel generado proceduralmente (9x9).*

Capítulo 5

Conclusión

Este proyecto ha servido como forma de instruirse desde cero en el uso de un motor gráfico de una forma autodidacta a través de contenido como vídeos, foros, tutoriales, manuales, etc. Otra preparación requerida ha sido la del lenguaje C# que comparte mucho parecido con Java y C. La herramienta Unity aporta unas bases y conceptos magníficos para adentrarse en el mundo del diseño y la programación de videojuegos, pero que también muestra un mundo lleno de complejidad y dedicación si se quiere alcanzar un producto agradable y bien construido.

Se ha experimentado el desarrollo de un pequeño juego independiente aplicando los roles de diseñador, programador y artista, adaptando conceptos totalmente distintos y en algunos casos desconocidos entre cada uno de ellos.

El concepto de generación procedural tiene un punto de vista muy complejo a la hora de ser adaptado en los juegos de plataformas, teniendo su mayor auge en juegos de mazmorras o salas donde este control es más eventual y específico en la escena, no en el nivel construido. Esta conclusión se basa en pura experimentación con títulos de estos géneros durante el desarrollo del proyecto.

El producto final que se ha logrado consigue generar una atmósfera retro gracias al estilo de arte pixelart que se combina con el género de plataformas de los años 80. Todo este conocimiento adquirido sirve de motivación de futuros proyectos y permite contemplar a Unity no solo como una herramienta para hacer videojuegos, sino como una herramienta de desarrollo para cualquier tipo de aplicaciones que dispongan de interfaz de usuario.

Desde un punto de visto de mercado, los proyectos de videojuegos independientes cada vez son más apoyados por plataformas como Kickstarter, que permite el apoyo de proyectos creativos de cualquier género o, Steam, que a través de su plataforma Greenlight la comunidad de usuarios permite votar proyectos independientes de interés y dar salida en mercado a estos que logran alcanzar una puntuación mínima.

Referente a la conclusión personal, el proyecto ha sido un trabajo muy elaborado y personal, lleno de técnicas y herramientas no tan cercanas a la carrera como tal, exceptuando la de programación, que he conseguido aprender y valorar como trabajo personal.

Concretando algunas de estas técnicas, la más costosa ha sido la animación de elementos mediante sprites y su edición con la herramienta Adobe Photoshop CC. En un principio todo el arte iba a ser con diseños propios pero requerían un tiempo y dedicación excesivos para lograr un trabajo aceptable.

Las mecánicas (saltar, correr, moverse, agacharse) necesitan ser pulidas para que se logre un efecto agradable y fluido con cualquier tipo de controlador, se tuvieron que modificar una gran cantidad de veces, parámetros como la gravedad, la fricción, el efecto rebote, etc, además de su posterior adaptación al autómata de movimiento con transiciones suaves.

Las decisiones de diseño han sido cruciales para lograr un proyecto adaptable a las limitaciones de tiempo y me han dado la capacidad de ver las limitaciones de recursos en algunos aspectos como la dimensionalidad, presupuesto y alcanzabilidad de los objetivos.

Como última nota, todos los conocimientos adquiridos me sirven como un planteamiento personal de la industria del videojuego y mis esfuerzos aplicados a ella que me gustaría seguir trabajando y perfeccionando.

Kure World

Capítulo 6

Trabajos futuros

Algunas ideas para mejorar el juego y apartado técnico que complementarían la experiencia jugable pero que por falta de medios o tiempo no se han incluido o no se han terminado.

- Mejorar e incluir el sistema de generación procedural de niveles que sea capaz de gestionar una dificultad adaptada.
- Mejorar el apartado multijugador local e incluir un apartado multijugador online.
- Mejorar el apartado artístico, sonoro y técnico.
- Diseñar e incluir un creador de niveles que facilite el diseño y creación de estos.
- Incluir un modo campaña con niveles encadenados que cuenten una historia.
- Introducir mecánicas como nadar, escalar y volar.
- Diseñar e incluir mas items, elementos, música y fondos.
- Diseñar una herramienta para compartir niveles.

Capítulo 7

Apéndice

7.1. Gestión del proyecto

Esta sección está dedicada a la gestión del proyecto, que define desde un breve análisis de las fases, la planificación que se ha seguido junto al diagrama estimado llevado a cabo y el presupuesto detallado con los costes invertidos para el desarrollo [4].

7.1.1. Descripción de fases del proyecto

El proyecto sigue un ciclo de vida evolutivo cíclico, resumido a continuación. Cabe destacar que el producto final no llegó a la fase de *Liberación* porque no se llegó a distribuir.

- **Diseño de concepto:** Fase inicial. Define el proyecto en grandes rasgos con una propuesta inicial, descripción de mecánicas, objetos, viabilidad, etc.
- **Pre-producción:** Diseño de especificaciones y funcionalidades, arte conceptual, idea visual y prueba de ideas.
- **Producción:** Implementación del proyecto.
- **Alfa/Beta:** Primeras fases de pruebas de la aplicación.
- **Liberación:** El proyecto se valida y empieza su distribución.
- **Actualizaciones:** Tras la liberación del producto este se puede actualizar o corregir si es necesario.

7.1.2. Planificación

La planificación seguida es de ciclo de vida en espiral, que ofrece flexibilidad para incluir requisitos según avanza el proyecto por falta de experiencia en el diseño. Se utilizó Git (software de control de versiones), en concreto la plataforma GitHub, para llevar una mejor notación de cambios y modificaciones. Se han desarrollado los siguientes módulos:

- **Aprendizaje y formación:** Etapa asignada al aprendizaje del motor Unity a base de tutoriales, vídeos explicativos, guías y consejos. También se ha dedicado tiempo al aprendizaje del lenguaje C#, las plataformas objetivo y apartados de diseño.
- **Documentación:** La documentación del proyecto empezó desde el día 0, anotando avances, aprendizaje y artículos de interés. Cabe destacar que era a tiempo parcial y la memoria se redactó en aproximadamente 3 meses después de finalizar el proyecto.
- **Desarrollo base:** Es un concepto en el que gira el juego, se basa en la carga de ficheros JSON y la instanciación de los elementos que componen el nivel leído. Es el más complejo de diseñar y su buen funcionamiento ha sido clave para el avance del proyecto.
- **Físicas:** Todo lo relacionado con las físicas del juego, en concreto la gravedad del mundo creado y las colisiones que se generan entre los distintos elementos disponibles.
- **Gestión de niveles:** Un apartado que integra la lectura JSON adaptándola con la interfaz del juego.
- **GUI:** (del inglés graphical user interface) Todo lo relacionado con la interfaz del juego y el HUD que muestra la información al usuario.
- **Lógica de juego:** Todo lo referente a mecánicas, objetivos y funcionalidades que cumples las entidades del juego.
- **Audio:** Inclusión de sonidos y efectos sonoros al juego.
- **Otros:** Este apartado recoge lo que se ha ido trabajando siempre en paralelo para su mejora, como el apartado artístico con la mejora e inclusión de nuevos sprites o el control del jugador que se ha ido adaptando y corrigiendo.

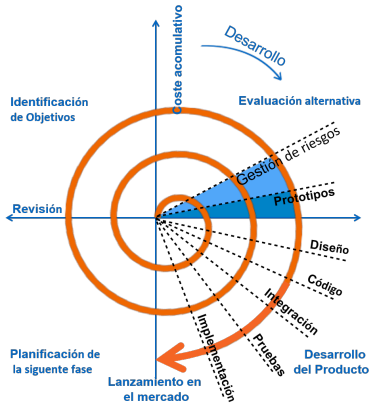

Figura 7.1: *Ciclo de modelo evolutivo en espiral.*
Fuente: `https://www.tutorialspoint.com`

7.1.3. Diagrama de Gantt

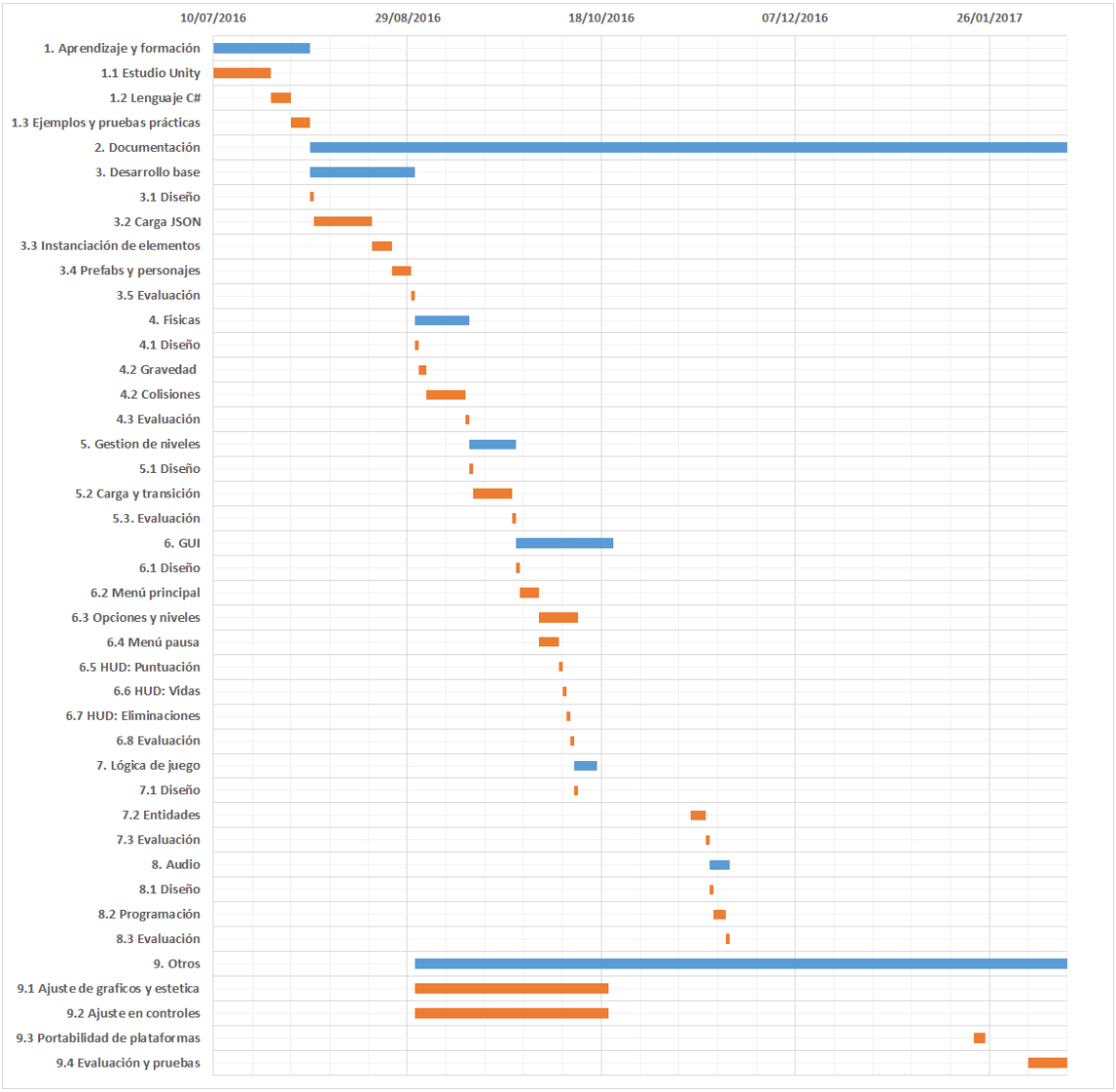

Figura 7.2: *Diagrama de Gantt.*

7.1.4. Diagrama de planificación

Tabla 7.1: Diagrama de planificación.

Etapas del proyecto	Duración estimada (días)	Inicio estimado (Día/Mes/Año)	Dedicación (Horas)
1. Aprendizaje y formación	25	10/07/2016	80
1.1 Estudio Unity	15	10/07/2016	55
1.2 Lenguaje C#	5	25/07/2016	20
1.3 Ejemplos y pruebas prácticas	5	30/07/2016	5
2. Documentación	300	04/08/2016	250
3. Desarrollo base	27	04/08/2016	54
3.1 Diseño	1	04/08/2016	4
3.2 Carga JSON	15	05/08/2016	30
3.3 Instanciación de elementos	5	20/08/2016	10
3.4 Prefabs y personajes	5	25/08/2016	5
3.5 Evaluación	1	30/08/2016	5
4. Físicas	14	31/08/2016	25
4.1 Diseño	1	31/08/2016	5
4.2 Gravedad	2	01/09/2016	5
4.2 Colisiones	10	03/09/2016	5
4.3 Evaluación	1	13/09/2016	10
5. Gestión de niveles	12	14/09/2016	16
5.1 Diseño	1	14/09/2016	4
5.2 Carga y transición	10	15/09/2016	6
5.3. Evaluación	1	25/09/2016	6
6. GUI	25	26/09/2016	40
6.1 Diseño	1	26/09/2016	5
6.2 Menú principal	5	27/09/2016	5
6.3 Opciones y niveles	10	02/10/2016	5
6.4 Menú pausa	5	02/10/2016	5
6.5 HUD: Puntuación	1	07/10/2016	5
6.6 HUD: Vidas	1	08/10/2016	5
6.7 HUD: Eliminaciones	1	09/10/2016	5
6.8 Evaluación	1	10/10/2016	5
7. Lógica de juego	6	11/10/2016	15
7.1 Diseño	1	11/10/2016	8
7.2 Entidades	4	10/11/2016	2
7.3 Evaluación	1	14/11/2016	5
8. Audio	5	15/11/2016	10
8.1 Diseño	1	15/11/2016	3
8.2 Programación	3	16/11/2016	5
8.3 Evaluación	1	19/11/2016	2
9. Otros	200	31/08/2016	65
9.1 Ajuste de gráficos y estética	50	31/08/2016	25
9.2 Ajuste en controles	50	31/08/2016	30
9.3 Portabilidad de plataformas	3	22/01/2017	5
9.4 Evaluación y pruebas	10	05/02/2017	5

Leyenda de planificación:

- **Duración estimada (días):** Es una estimación en días desde el inicio del proceso a su finalización, cabe destacar que el trabajo realizado no era a tiempo completo y había días en los que solo se retocaban detalles o anotaciones.
- **Inicio estimado (Día/Mes/Año):** Fecha estimada del inicio del proceso.
- **Dedicación (Horas):** Dedicación aproximada en horas del proceso. Cabe destacar que no eran continuas ni a tiempo completo.

7.2. Presupuesto

Este apartado desglosa el presupuesto que tiene el proyecto. Personal, Seguridad Social, medios materiales y amortización así como costo total del trabajo se verán reflejados en este apartado. Cabe destacar que el proyecto fue desarrollado por una única persona y estos datos son solo una mera aproximación real con más personal con salarios obtenidos del Boletín Oficial del Estado según grupos profesionales y nivel de funciones [4].

- **Autor:** Daniel Jerez Garrido
- **Departamento:** Informática - Computación
- **Descripción:**
  - Generación de contenido procedural en videojuegos basados en Unity.
  - Duración estimada 4 meses a tiempo no completo.
  - IVA 21 %.
- **Presupuesto total del proyecto (Euros):**
  9.502 €
- **Desglose presupuesto (costes directos):**
  - **Personal:**

Tabla 7.2: Costes de personal.

Perfil	Precio/Año	Meses	Coste Final
Analista programador	21.800 €	2	3.633,33 €
Programador	18.100 €	2	3.016,67 €
Testeador	13.000 €	1	1.083,33 €
			7.733,33 €

- **Material y licencias:**
  - **Hardware de desarrollo y pruebas**
    - **Procesador:** Intel Core i7 4770 3400 MHz
    - **Tarjeta Gráfica:** NVIDIA GeForce GTX 660
    - **Memoria RAM:** 8192 MB (DDR3-1333 SDRAM)
    - **Disco Duro HDD:** 1TB
    - **Disco Duro SDD:** 120GB
    - **Caja:** COOLER MASTER SILENCIO 352
    - **Placa:** ASUSTeK COMPUTER INC. Z97M-PLUS
    - **Fuente:** 750 W TOOQ ECOPOWER
  - **Pruebas en Android:** Xiaomi Mi4c
  - **Software utilizado (Licencias sin coste no incluidas en presupuesto)**
    - Unity 5.6.1f1 (64-bit) (Licencia personal) (`https://unity3d.com/es`)
    - Adobe Photoshop CC (`www.adobe.com/es/photoshop`)
    - Sharelatex (`https://es.sharelatex.com`)
    - Microsoft Office 365 (`https://www.microsoft.com`)
    - GitKraken (`https://www.gitkraken.com`)
    - GitHub (`https://github.com/danijerez`)
    - itch.io (`https://danijerez.itch.io/kure-world`)
    - Java JDK 8 (`https://www.oracle.com`)
    - draw.io (`https://www.draw.io`)
    - DirectX 11/12
    - Microsoft Visual Studio 2017 (`https://www.visualstudio.com`)
    - Android SDK (`https://developer.android.com/studio/index.html`)
    - Sublime Text 3 (`https://www.sublimetext.com`)
    - Elementary OS - Loki (`https://elementary.io/es`)
    - Mac OS X - El Capitan (`https://www.apple.com/es`)
    - Lineage OS (Android 7.1.2) (`https://lineageos.org`)
    - Windows 10 x64 (Coste de licencia incluida en el ordenador)

Tabla 7.3: Costes de material.

Descripción	Coste	% Uso	Dedicación	Periodo de deprecación	Coste imputable
Ordenador	840,00 €	100	4	60	56,00 €
Xiaomi Mi4c	250,00 €	100	2	60	8,33 €
					64,33 €

Tabla 7.4: Costes en licencias.

Descripción	Coste	% Uso	Dedicación	Periodo de deprecación	Coste imputable
Office 365	149,00 €	100	4	60	9,93 €
Photoshop CC	80,00 €	100	4	60	5,33 €
					15,27 €

Fórmula de cálculo de la amortización $= \frac{A}{B} * C * D$

- **A** = nº de meses desde la fecha de facturación en que el equipo es utilizado
- **B** = Periodo de deprecación (60 meses)
- **C** = Coste del equipo (sin IVA)
- **D** = % de uso dedicado a proyecto (100 % normalmente)

- **Costes indirectos:**

Tabla 7.5: Costes indirectos.

Descripción	Coste
Transporte	40,00 €
	40,00 €

- **Resumen:**

Tabla 7.6: Resumen de costes.

Costes totales	Presupuestos totales
Personal	7.733,33 €
Amortización (Material)	64,33 €
Amortización (Licencias)	15,27 €
Indirectos	40,00 €
IVA	1.649,12 €
	9.502,05 €

7.3. Palabras clave

Esta sección recoge los términos mas técnicos mencionados.

- **Videojuego:** Juego electrónico que se visualiza por medio de una pantalla.
- **Plataformas:** Género de videojuegos que se caracteriza por controlar a un personaje a lo largo de un nivel lleno de obstáculos, plataformas y enemigos con distintas mecánicas y cuyo objetivo principal es completarlo cumpliendo unos requisitos (tiempo, vidas, meta, etc). Las principales mecánicas de las que dispone el género son caminar, saltar, correr, escalar, disparar, nadar, volar, etc. La cámara de los plataformas 2D suele ser ortográfica y su desplazamiento es en scroll lateral (derecha e izquierda) aunque algunos títulos incluyen verticalidad (arriba y abajo).
- **Unity:** Es un motor de videojuegos multiplataforma creado por Unity Technologies disponible como plataforma de desarrollo para Microsoft Windows, OS X y Linux.
- **JSON:** Acrónimo de JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos.
- **Procedural:** Es un método de creación de datos con algoritmos en lugar de forma manual.
- **Sprite:** Mapa de bits que representan una imagen.
- **HUD:** (head-up display) Es una pantalla transparente que presenta información al usuario y esta superpuesta al contenido mostrado.
- **Collider:** Componente que sirve para detectar colisiones dentro de un objeto.
- **Rigidbody:** Componente que permite el comportamiento físico para un objeto.
- **Roguelike:** Es un género de videojuegos donde cada partida se empieza desde el principio, suele ser aleatorio y tener una dificultad elevada.
- **Indie:** Término usado para los videojuegos de desarrollo independiente.
- **Prefab:** Objeto prefabricado con componentes y propiedades especificas.

7.4. Manual de usuario

A continuación se define la información necesaria para que el usuario sea capaz de ejecutar correctamente el juego.

7.4.1. Requisitos de sistema

Requisitos hardware:

- Procesador 1.5 Ghz
- RAM: 2 Gb
- Almacenamiento: 200 MB

S.O. compatibles:

- Windows 7/8/8.1/10
- Distribución de Linux kernel 4 o superior
- Mac OS X
- Android 4 o superior.

7.4.2. Instalación

Se necesita conexión a internet para descargar el juego. El proyecto se encuentra alojado en la plataforma itch.io (`https://danijerez.itch.io/kure-world`) contraseña: tfgdanieljerez. En esta página aparece un breve resumen del juego, unas capturas de pantalla y los enlaces de las plataformas en las que está disponible.

Se necesita un descompresor de ficheros .zip para las plataformas de windows, mac os x y linux, en android solo es necesario habilitar la opción de instalación de aplicaciones de origen desconocido.

Figura 7.3: *S.O. disponibles*

7.4.3. Manual de uso

- Tras la instalación, lanzar el ejecutable. La carpeta "levels"se autogenera si no existe, es donde se almacenan los niveles y la creación de nuevos ha de estar incluida en esta carpeta.

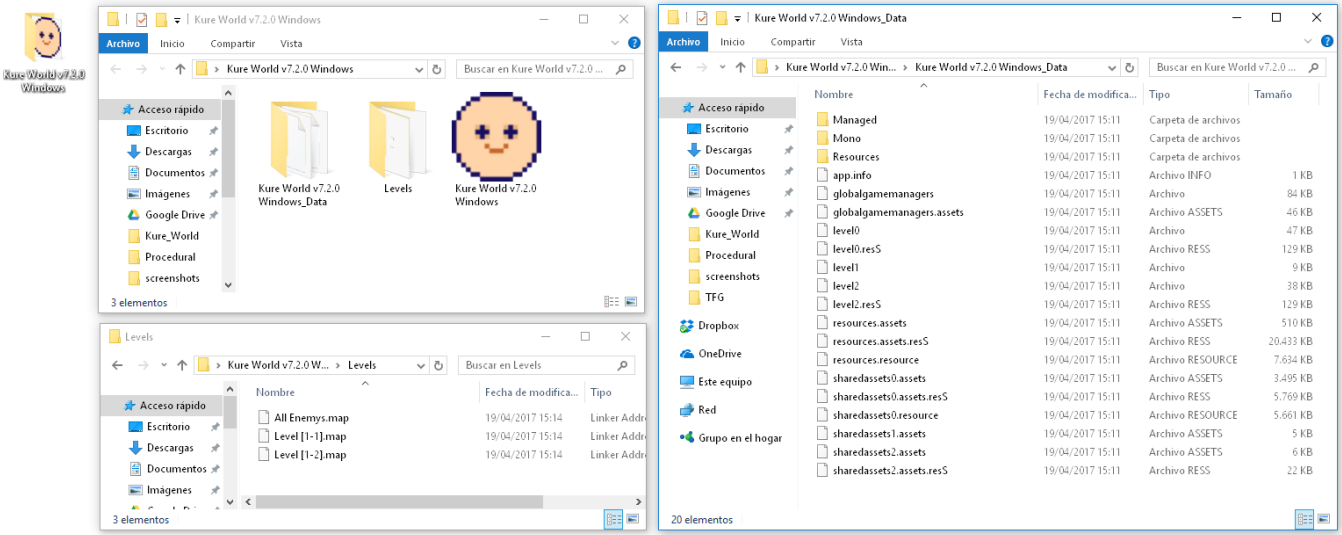

Figura 7.4: *Estructura de carpetas y ficheros (Windows)*

- La pantalla inicial es de ajustes previos (no disponible en android) donde se puede configurar la resolución de pantalla, el modo pantalla completa o ventana y los controles del juego.

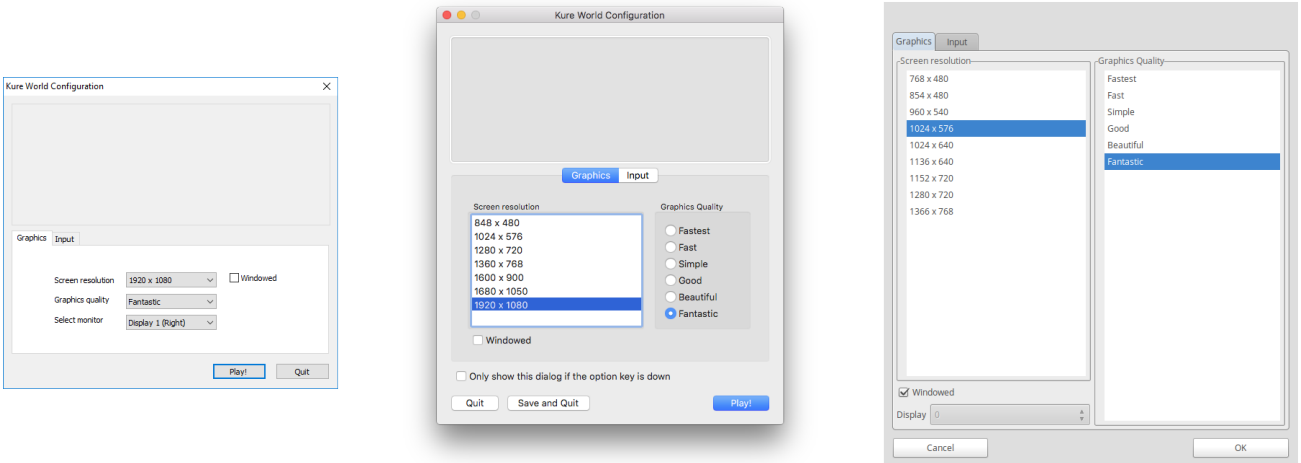

Figura 7.5: *Interfaz ajustes previos (Windows, Mac OS X, Linux)*

- Tras pulsar Play! se lanza el juego mostrando inicialmente el logo de Unity y posteriormente el menú principal donde se puede elegir el nivel, ajustar opciones o salir del juego. El botón *Procedural generation test level* permite el acceso al menu de las pruebas de generación procedural.

Figura 7.6: *Menú principal*

- Las opciones se muestran al pulsar el icono de tres líneas paralelas. Para ocultarlas solo hace falta hacer click en la zona oscurecida.

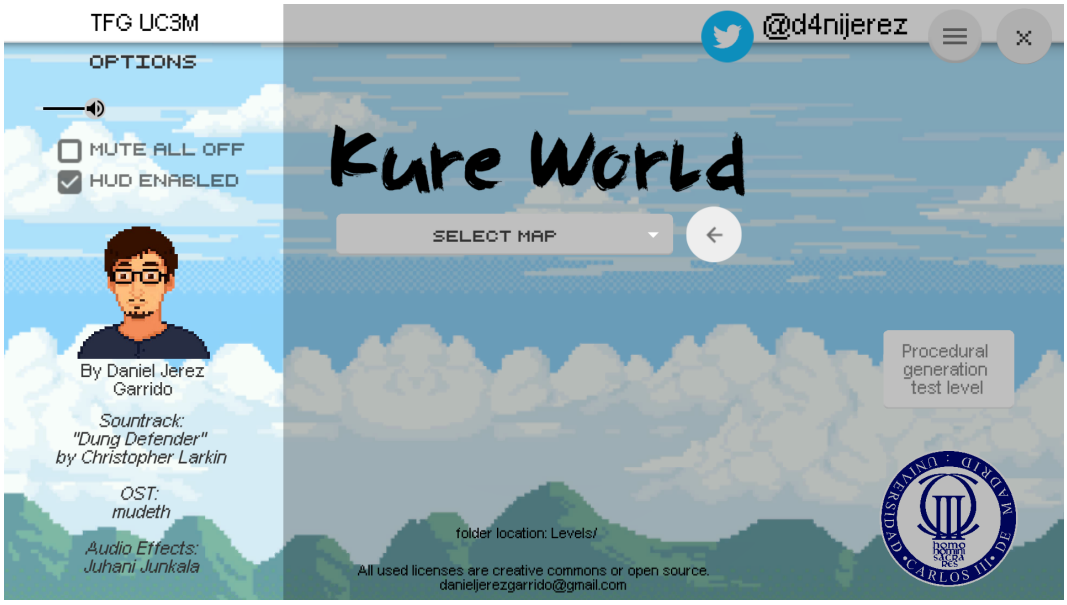

Figura 7.7: *Opciones del menú*

- Cuando se elige nivel, aparece una pantalla de información que permite al usuario saber las vidas que tiene, el tiempo y el autor. Posteriormente empieza el nivel.

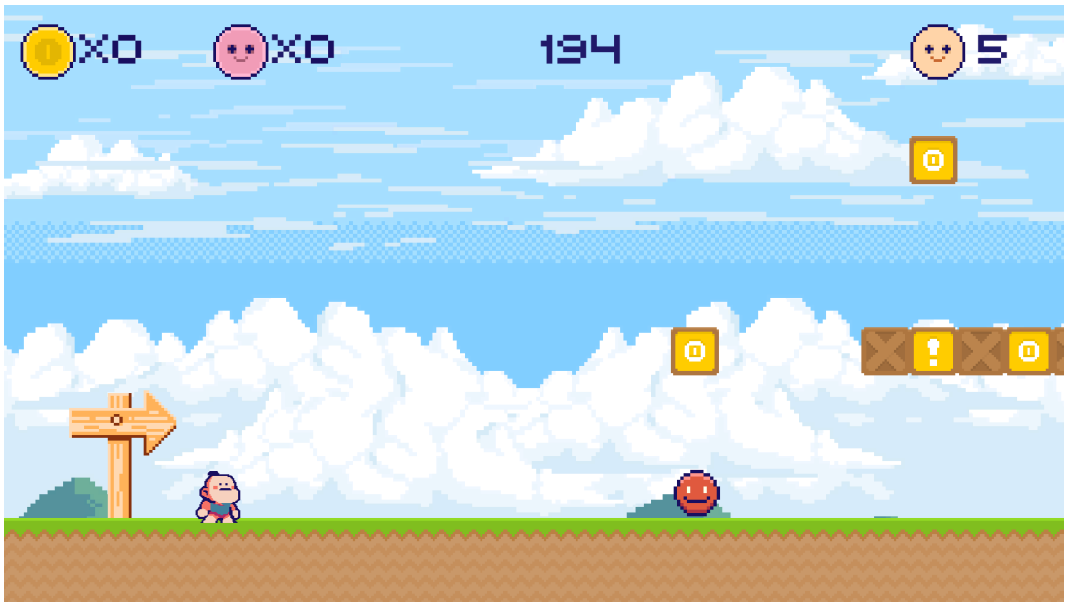

Figura 7.8: *Interfaz del juego en un nivel*

- Al pulsar el botón de pausa en un nivel, el juego se pausa y aparecen las opciones de reintentar, salir, mutear el sonido, ajustar el sonido y ocultar el HUD.

Figura 7.9: *Interfaz menú de pausa*

7.5. Códigos de elementos

Cada elemento (Prefab) disponible y programado en el juego esta catalogado en las secciones: Construcción (Buildings), Poderes (Powers UP), Personaje (Character) y Enemigos (Enemies). La disposición y colección esta estructurada de tal forma que sea fácil integrar nuevos elementos en futuras actualizaciones. A continuación se muestran los disponibles en la versión final. Leyenda de tabla (ID / Sprite / Nombre).

7.5.1. Construcción (Buildings)

1		Brick
2		Brick Blue
3		Box power
4		Box coin
5		Box star
6		Box null
7		Block
8		Floor 0
9		Floor 1
10		Flag1

11		Flag2
12		Arrow
13		Coin
14		Cloud
15		Spike
16		Castle
17		Conveyor 1 ←
18		Conveyor 2 →
19		Pipeline 1

20		Pipeline 2
21		Pipeline 3
22		Pipeline 4
23		Floor 2
24		Floor 3

Figura 7.10: *Códigos - Construcciones*

7.5.2. Poderes (Powers UP)

97		Power up
98		Flower fire
99		Star

Figura 7.11: *Códigos - Poderes*

7.5.3. Personaje (Character)

70		Phase 1
72		Phase 2
74		Phase 3

Figura 7.12: *Códigos - Personaje*

7.5.4. Enemigos (Enemies)

76		slime
77		Canon bean 1
78		Canon bean 2

Figura 7.13: *Códigos - Enemigos*

Capítulo 8

Summary

This chapter corresponds to the summary in English that must be included in the computer engineering degree of the 2011 plan. Its structure is composed of introduction, objectives, results (final product and how it has been built) and conclusions.

Abstract

This project is based on the development of a fully functional 2-dimensional platform game for Windows, Linux, Mac Os X and Android platforms using the Unity graphics engine.

Techniques of construction of automated scenarios are realized through loading of JSON files edited by the user. The files contain all the necessary information that the game needs for the staging (block codes, enemies, background, music, player start and goal). The level is structured in a coded two-dimensional matrix that the game traverses and instantiates each element with its respective properties programmed. In addition, procedural generation techniques are studied as an alternative to the manual creation of mentioned levels.

Keywords

Videogame ; Platforms ; Unity ; JSON ; Procedural

8.1. Introduction

The project is called *Kure World*, it consists of the development of a fully functional video game of the sort of side scroll platforms in 2 dimensions with orthographic camera using for its development the graphic engine Unity 3D and the programming language C#.
Two lines of work have been followed.

- Development and functionality of the video game. Learning the use of the Unity tool from scratch with the objective of learning the development phases of a software project and the construction of a game that complies with a mechanics and a goal, as well as a user-friendly interface.
- Research and implementation of methods of loading files for automation in the construction of levels that follow a hierarchy and codification raised, besides a study of procedural generation that allows the automatic generation of these.

8.2. Objetives

The main objectives of the project can be defined in the following:

- Perform a 2D platform video game with the Unity tool and make it fully functional for the Windows, Mac OS X, Linux and Android platforms.
- To perform a load of levels contained in JSON files containing an array with the encoded elements in addition to other attributes such as background, music, number of lives, author of level and as optional part the next level.
- Design an interface that allows manipulating the video game, being able to choose between the list of available levels at that moment.
- Perform a study of a possible procedural generation of levels that is compatible with this game structure.

8.3. Results

This section describes the results achieved and the process followed.

8.3.1. JSON files and level loading

The main idea of the project was to generate a basic load of a text file and that the program was able to adapt that read data and build it in the game. The flat texts could be configured as a square matrix where each element (characters) can be dimensioned in the code. The initial versions took two alphanumeric characters, allowing a large number of combinations. The code focused on reading the content line by line and going by cutting and storing every two characters in an array to later make a comparison of each code and build them.

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000B3000000B
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
000000000000M1000000C0C0C0C000000000000001
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
```

Figura 8.1: *Example level txt.*

The main design errors of this method were the inability to enter more information into the text file itself, leaving options such as life and time not being set by the creator of the level if not by a player from the menu Options.

The alternatives were to change the type of file, the options that were raised were:

- Binary files, but they could not be edited with a text editor and they lost that feature.
- Enter additional lines in the text file, but it was somewhat crude and difficult to set empty parameters or not wanted to be included in the level.
- Use an information exchange format such as XML, YAML or JSON since they can be easily edited by any editor and use the variable concept to store information.

The final decision was JSON (JavaScript Object Notation) for prior knowledge in Python and the ability to structure square matrices that are perfect for representing a 2d side scroll mapping. This type of structure allows you to enter new fields to store the number of lives, the time available to pass the level, the author of the level that is displayed before starting it, the background music, the background image and a possible next level that will be loaded at the end of the current one successfully.

```
{
"author":"dani",
"lives":05,
"time":400,
"background":01,
"music":03,
"next":"single [1-2]",
"matrix":
[
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,04,00,00,00,01,03,01,00],
        [00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,12,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00],
        [00,00,00,00,70,00,00,00,00,00,00,00,00,00,00,00,00,00,76,00,00],
        [09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09,09],
        [08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08],
        [08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08,08]
]
}
```

Figura 8.2: *Example level with structure JSON.*

For the implementation, a free library was used as part of the Unity assets offered to the community. Some of the parameters for reading the matrix had to be modified, the library is adapted for the use of one-dimensional vectors and has some difficulties in reading higher dimensions. The library is called JSONObject and its link is as follows. `https://www.assetstore.unity3d.com/en/#!/content/710`

8.3.2. Instantiation of elements

As discussed in the previous section, once the structure of the JSON file has been read and the coded array of the elements that compose the map stored, the process necessary to generate it within the game is through an instantiation associated with a prefab previously designed for that element. The codes are numeric without limit, property obtained thanks to the files JSON, which establishes to be able to integrate to the game infinite elements for the creation of levels.

The code implementation is based on a switch that identifies the id of the element and locates the prefab path. This prefab is instantiated in the coordinate corresponding to the matrix starting at the point (0, 0) and taking displacements according to the size of the instantiated element.

8.3.3. Component coding and composition

All instantiated components are designed and constructed in a type of object called prefab. These are mostly composed of 4 characteristic subcomponents located in the Inspector.

- **Transform:** Contains the positioning information of the object (Position, Rotation and Scale).
- **Sprite renderer:** Contains the image associated with the element and some configurable parameters such as material, orientation and drawing mode.
- **Collider 2D:** It allows to associate to the element a type of collision, usually they are square or circular and if a rigidbody component is included the physical property and access to the movement controller is obtained.
- **Script:** Code file that includes specific instructions for the prefab. It is always composed of two main functions. Start, which would be a kind of constructor, is launched in the creation of the object and configures all the parameters necessary for its operation, and Update, function that is launched in each clock tick and allows to introduce continuity mechanics.

The prefabs are classified in 5 sections so that they can be conveniently located.

- **Building:** All types of constructions, from soil types, blocks, boxes, flags, coins, etc.
- **Character:** Characters available in their different phases of evolution.
- **Danger:** Items that damage or are thrown by the player.
- **Enemies:** All enemies available.
- **Powers:** Elements that allow the character to evolve, powers up, immunity stars and fire flowers.

Figura 8.3: *Prefabs available grouped by sections.*

8.3.4. Character Controller

The input (Character Controller) is divided into horizontal movement, running, jumping, shooting and squatting. This whole process is programmed in the script added to the prefab of the character which is endowed with a circle of collisions (Circle Collider) that detects when it has something under its feet, like a block to lie on the ground, an enemy to crush it or a Block that hit. Each action of the controller that is performed has associated an animation of sprites and a certain sound. The specific functions of the controller and a fragment of its implementation are defined below.

- **Horizontal motion:** is handled by the 2D Rigidbody component within the prefab of the character. This component captures when a movement key is pressed and adds a horizontal force in that direction to move the character. The designated keys are WASD and the direction keys, in addition in Android devices appears a Joystick that adapts this control.

- **Run:** Exactly the same as the horizontal movement but the scrolling is greater when you press the F key or the B button on Android.

- **Skip:** Pressing the jump key exerts a vertical upward displacement force. The jump can only be executed if the character has touched the ground previously. The assigned keys are the space button and the A button on Android.

- **Shoot:** The shooting option is only available after touching a fire flower. Pressing the trigger key generates a ball with a certain force that comes out in front of the character. The assigned keys are F and B on Android.

- **Crouching:** When the crouch key is pressed, the character immobilizes and occupies a block of high without depending on the state in which it is, the only condition is that it is perched in a block.

8.3.5. Sprites, Sounds and Interface

A Sprite is an image or set of images that allows visual representation of elements of the game. The characteristic properties of the Sprite are that they are essentially for games in 2 dimensions and between the set used have a similar resolution that allows integrating with each other. The sprites database that has been used in the project is «`https://opengameart.org` », which has open source material created by your community, specifically the one chosen follows a pixelart themed (use of very low resolutions, few colors and animations with few frames). Specific sources are:

- **Blocks, floors and boxes:** `https://opengameart.org/content/platformer-art-pixel-redux`

- **Enemies:** `https://opengameart.org/content/platformer-baddies`

- **Character:** `https://opengameart.org/content/superpowers-assets-characters`

- **Own Textures**: All of the sprites have been previously retouched and modified with the Adobe Photoshop tool, some of the elements used are in-house.

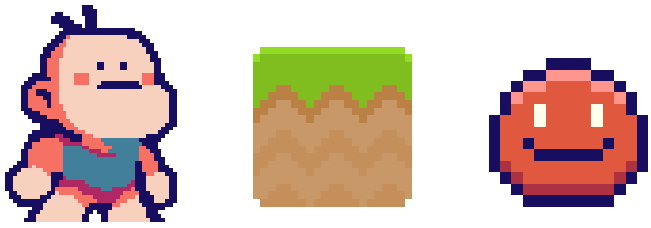

Figura 8.4: *Example of Final Sprites. (Character/Ground/Enemy)*

The audio effects used in the project (Jump, Shoot, etc) are subtracted from (`https://opengameart.org/content/512-sound-effects-8-bit-style`) which has a wide Variety of sounds to 8 bits of free use.

Background music is created by author Mudeth (`https://mudeth.bandcamp.com/album/antibirth-ost`), he was asked permission to use the compositions in the project. The music is part of the OST of the Antibirth mod of the game The Binding of Isaac - Afterbirth.

The interface is designed with the Unity engine itself and a Material UI library (`https://www.assetstore.unity3d.com/en/#!/content/51870`) to give a nice design material effect for the user (clean, tidy and simple).

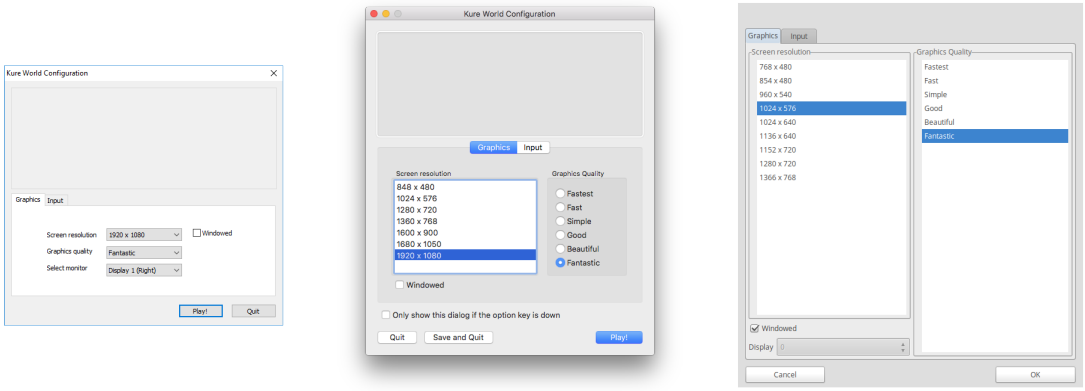

Figura 8.5: *Interface pre-settings (Windows, Mac OS X, Linux)*

Figura 8.6: *Main menu interface*

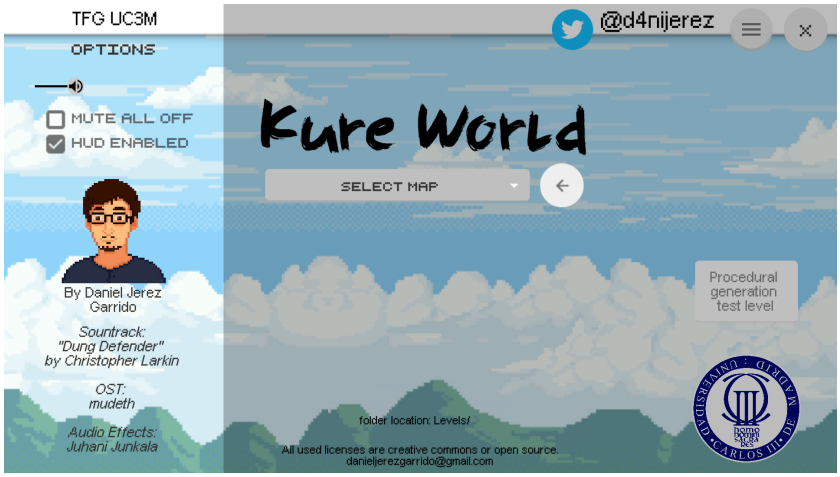

Figura 8.7: *Interface Options*

Figura 8.8: *Interface pre-level information*

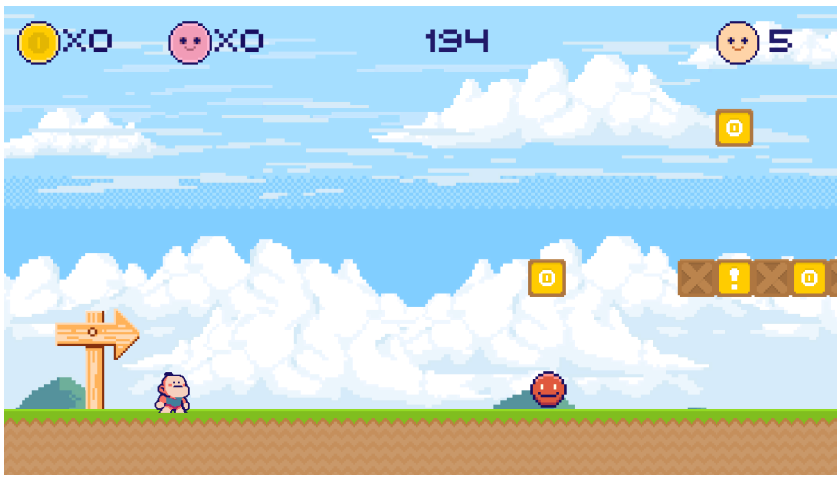

Figura 8.9: *Game interface on one level*

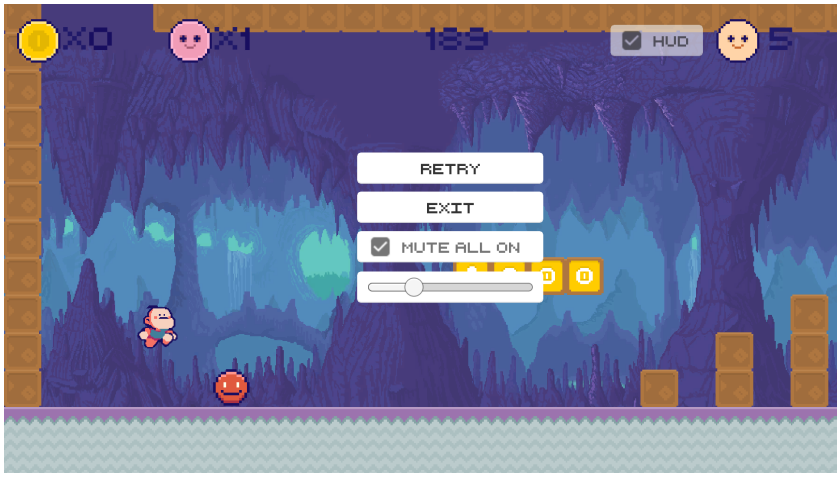

Figura 8.10: *Pause menu interface*

8.3.6. Animation automaton and skeleton elements

Prefabs that require a series of animations in specific situations need an automaton that manages the current state and the transition to a new state. Specifically, the character's automaton is the most elaborate, its initial default state is stop and when its velocity is greater than 0.01 it travels to run or jump if the jump key has been pressed and the Y axis has been modified. From run you can run (turbo) and run jump (turbo jump). From jump it can only fall or stop again. Some decisions can be taken from any state, shoot or win.

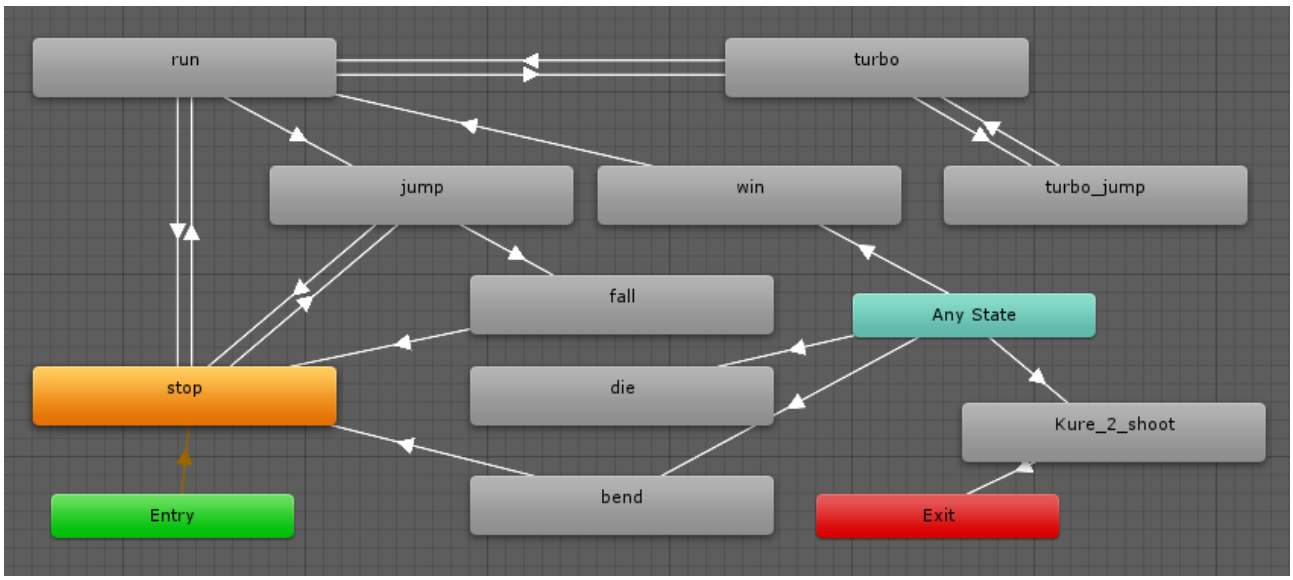

Figura 8.11: *Character movement automaton*

The objects also include a skeleton that allows them to detect these situations. The character has two collider, the largest is the body, which detects collisions with boxes and enemies and the smallest, which are the feet, serves to detect when it is touching ground. The enemies IA is simple, they only advance if the character is at a certain distance, they collide with something change of sense (this is detected with the circular colliders placed on the sides), to know if they are crushed they have a square collider In the upper body. The box also has two square collides, one to know when the character is posed above and another when you press it from below. The rest of the elements also have skeleton similar to those explained.

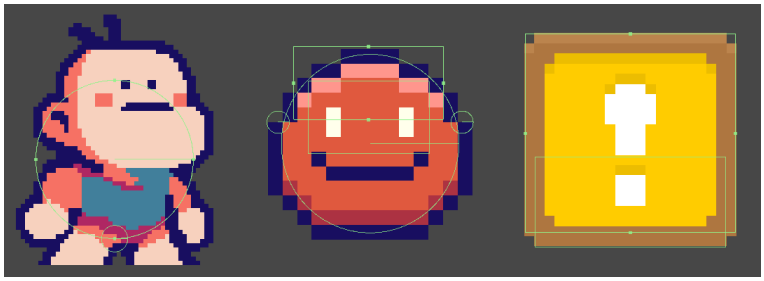

Figura 8.12: *Colliders that make up the character, enemy and box*

8.3.7. Study and testing of procedural generation

For the tests and implementation of a procedural algorithm we looked for types of games similar to the objective of the project. From a current point of view, there is no great variety of these that facilitate your code or algorithm, and the final quality of result is not expected due to the complexity that involves combining so many elements to almost random decisions.

The three properties that a procedural algorithm must meet for the generation of a level are:

- **Feasibility:** A level has to have beginning and end.
- **Design:** Must be an interesting and well-built design.
- **Difficulty:** Adaptable to the particular player who is playing.

Each one separately is easy to meet, but the combination of the three is a great difficulty in perfection. The reference games that fit these ideas within the genre 2d platforms are few and note that they are all indies. Taking these ideas the starting point was search and experimentation with Spelunky's algorithm, because of the mentioned games is the only one that has the code released. The main difficulties are that the game is built with the GameMaker engine and without license of the program it is not possible to visualize it besides that the programming language is different and its translation is expensive and complex. The second option is to test adaptations of this algorithm that users have done in unity, but the results are not faithful to the original and are based on random terrain bounded by a mesh of matrices that do not provide any challenge to the player because the obstacles are minimal and there are no other elements as enemies.

Figura 8.13: *Level generated with cellular automaton*
Source: `https://unity3d.com/es/learn/tutorials/projects/procedural-cave-generation-tutorial/cellular-automata?playlist=17153`

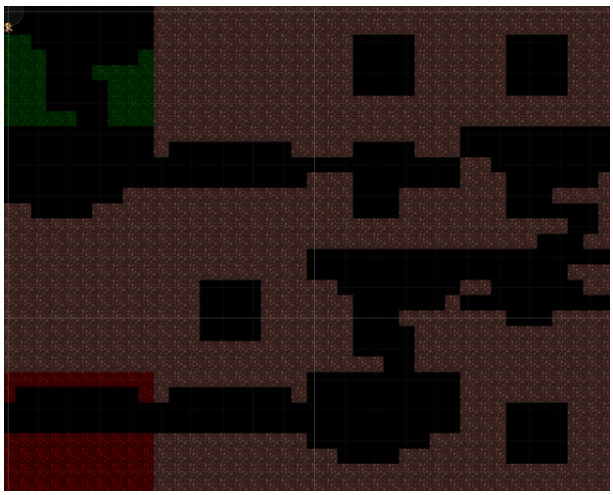

Figura 8.14: *Generated level with Spelunky algorithm adapted in unity (Matrix green start, final red matrix).*

Source: `https://github.com/gholaday/Procedural-Level-Generation`

Subsequent tests were from algorithms invented or created by users. Usually none was especially good, because they are really complex methods and suffer too many deficiencies of continuity besides that they were not similar to the aesthetics of the game. An algorithm was modified (Source: `https://github.com/stuartlong/chocolatebox`), which divides the mapping into sections that include obstacles such as precipices or elevations depending on a seed affected by a parameter of increasing difficulty as you passed the levels.

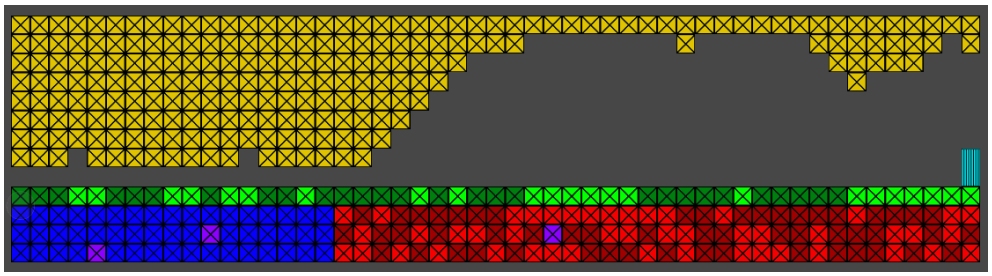

Figura 8.15: *Procedural level (1 section and low difficulty)*

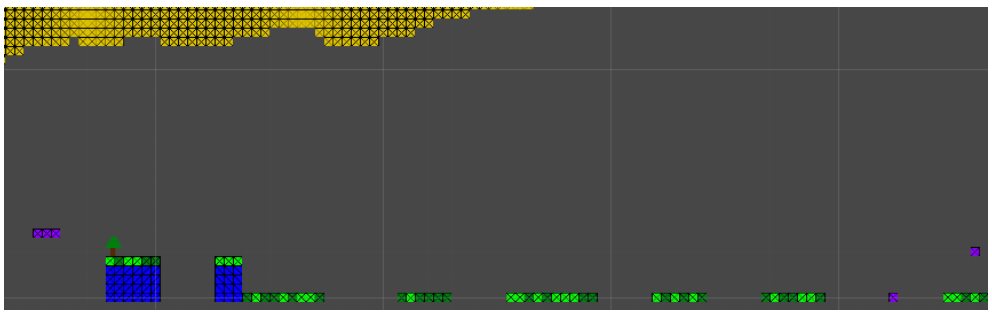

Figura 8.16: *Fragment of procedural level (3 sections and high difficulty)*

Although the results were better than expected, the coding was in a version of Unity inferior to the one of the project, the code had a great amount of failures that made it unstable and the integration to the game proved impossible since the prefabs were not compatible, the physical ones were posed in another way and the dimensionality was superior what showed an immense scene and a diminutive personage. The algorithm chosen to implement the project was the modified Spelunky, explained in detail below.

8.3.8. Implemented procedural generation

The implemented procedural generation algorithm is based on the modified project (`https://github.com/gholaday/Procedural-Level-Generation`).

The algorithm performs a scenario formed by predefined arrays. Each matrix is formed by two elements (0, 1), 0 is for empty voids and 1 is for elements. In turn these matrices are classified into 0, 1, 2 and 3.

- **0:** Matrix without hollows or with hollows interior but without roads.
- **1:** Matrix with horizontal paths.
- **2:** Matrix with vertical paths.
- **3:** Matrix with upper and horizontal paths.

```
rooms_0.Add(new StringArray(
    new string[]{
        "1111111111",
        "1111111111",
        "1110000111",
        "1110000111",
        "1110000111",
        "1110000111",
        "1111111111",
        "1111111111"
    }
));
rooms_0.Add(new StringArray(
    new string[]{
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111"
    }
));
```

```
rooms_1.Add(new StringArray(
    new string[]{
        "1111111111",
        "0000110000",
        "0000000000",
        "0000000000",
        "1111111111",
        "1111111111",
        "1111111111",
        "1111111111"
    }
));
rooms_1.Add(new StringArray(
    new string[]{
        "1111111111",
        "1110000100",
        "0000000000",
        "0000000000",
        "1110001110",
        "1110001111",
        "1111111111",
        "1111111111"
    }
));
```

```
rooms_2.Add(new StringArray(
    new string[]{
        "0000000000",
        "0000000000",
        "1100000000",
        "1110000001",
        "1110001111",
        "1110000111",
        "1110000111",
        "1111100111"
    }
));
rooms_2.Add(new StringArray(
    new string[]{
        "0000000000",
        "0000000000",
        "1100000000",
        "1110001111",
        "1110000000",
        "1110000000",
        "1111111001",
        "1111100001"
    }
));
```

```
rooms_3.Add(new StringArray(
    new string[]{
        "0000000000",
        "0100000111",
        "0000000000",
        "0000000000",
        "1110001111",
        "1110011111",
        "1111111111",
        "1111111111"
    }
));
rooms_3.Add(new StringArray(
    new string[]{
        "0000000000",
        "0000000000",
        "0000000000",
        "0000000000",
        "0000000011",
        "1100001111",
        "1111111111",
        "1111111111"
    }
));
```

Figura 8.17: *Example of arrays predefined by sections.*

The set of the matrices in turn form an upper matrix whose specifications were set from 5x5 to 9x9 variable ranges and randomly chosen that define the size of the level. This set of matrices generates a path that has an array as the start and a final matrix, marked in red, as the goal of the level.

The choice of matrices is random, a start position is chosen and a matrix 2 or 3 is placed. Matrices 1 can only be on sides, matrices 2 in lower positions and matrices 3 in both, this way it is always generated A possible path variable in length. The remaining free positions are completed with matrices 0 which serve to fill the voids of the matrix.

When the upper matrix is defined it is iterated on it and each element is instantiated. If the element is a 1, a prefab of terrain (block, box, ground, etc.) is loaded, if it is a zero (void free) a prefab is empt emptied, enemy, box, coin, etc. The percentages of these instantiations are variable and random parameters and increase the difficulty and the final result of the level.

Since this type of level generation is experimental, an access button was included in the main menu as test. Once inside the menu, the parameters life, music, time, background and size of the level for later generation are selected by pressing the Go! Button.

Figura 8.18: *Optional menu of procedural level generation tests.*

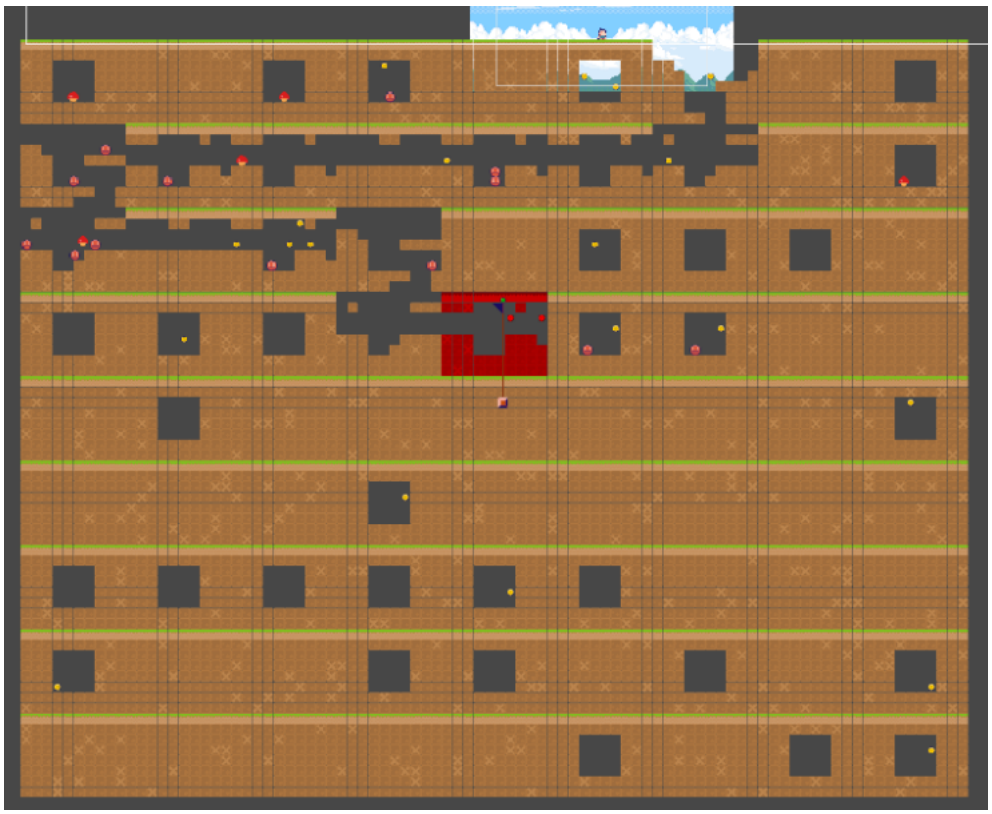

Figura 8.19: *Example of level generated procedurally (9x9).*

8.4. Conclusions

This project has served as a way to learn from scratch in the use of a graphic engine in an autodidact way through content such as videos, forums, tutorials, manuals, etc. Another required preparation has been the language C# which shares much like Java and C. The Unity tool provides a great foundation and concepts to enter the world of video game design and programming, but also shows a world full of complexity and dedication if you want to achieve a nice and well-built product.

We have experimented with the development of a small independent game applying the roles of designer, programmer and artist, adapting totally different concepts and in some cases unknown between each one.

The concept of procedural generation has a very complex point of view when it comes to being adapted in platform games, having its greatest boom in dungeon games or rooms where this control is more eventual and specific in the scene, not in the level built. This conclusion is based on pure experimentation with titles of these genres during the development of the project.

The final product that has been achieved manages to generate a retro atmosphere thanks to the pixelart style of art that is combined with the platforms genre of the 80's. All this knowledge acquired serves as a motivation for future projects and allows to contemplate Unity not only as a tool for making games, if not as a development tool for any type of applications that have an interface.

From a market point of view, independent video games projects are increasingly supported by platforms such as Kickstarter, which allows the support of creative projects of any genre or, Steam, through its Greenlight platform the user community allows to vote independent projects of interest and give out in the market those who achieve a minimum score.

Regarding the personal conclusion, the project has been a very elaborate and personal work, full of techniques and tools not so close to the career as such, except for the programming, which I have managed to learn and value as personal work.

Concreting some of these techniques, the most expensive one has been the animation of elements using sprites and their editing with the tool Adobe Photoshop CC. In the a beginning all the art was going to be with own designs but they required an excessive time and dedication to obtain an acceptable work.

The mechanics (jumping, running, moving, crouching) need to be polished to achieve a pleasant and fluid effect with any type of controller, had to be modified a lot of times, parameters such as gravity, friction, rebound effect, etc, besides its later adaptation to the automaton of movement with smooth transitions.

Design decisions have been crucial in achieving a project adaptable to time constraints and have given me the ability to see resource constraints in some aspects such as dimensionality, budget and achievability of the objectives.

As a final note, all the knowledge acquired serves me as a personal approach of the video game industry and my efforts applied to it that I would like to continue working and improving.

Kure World

Bibliografía

[1] C. Belli, Simone y López, "Breve historia de los videojuegos.athenea digital," *http://psicologiasocial.uab.es/athenea/index.php/atheneaDigital/article/view/570*, 2008.

[2] F. de informática de Barcelona, "Historia de los videojuegos," *https://www.fib.upc.edu/retro-informatica/historia/videojocs.html*, 2011.

[3] J. P. Donate, "Manual de creación de videojuego con unity 3d," *TFG*, pp. 14–153, 2012.

[4] M. A. Marquez Martinez, "Diseño e implementacion de un juego para smartphones con android," *TFG*, pp. 100–110, 2014.

[5] A. M. Mora, "Applications of evolutionary computation," *https://link.springer.com/chapter/10.1007/978-3-319-16549-3_25*, 2015.

[6] C. Pedersen, "Modeling player experience in super mario bros," *http://ieeexplore.ieee.org/abstract/document/5286482/?reload=true*, 2009.

[7] Roy, "Historia de los videojuegos: El origen y los inicios," *http://www.otakufreaks.com/historia-de-los-videojuegos-el-origen-y-los-inicios/*, 2011.

[8] J. V. Ryzin, "Space shuttle re-entry," *http://web.archive.org/web/20161106195319/http://thedoteaters.com:80/*, 1991.

[9] J. Schaal, "Procedural terrain generation. a case study from the game industry," *https://link.springer.com/chapter/10.1007/978-3-319-53088-8_8*, 2017.

[10] J. Tremblay, "I can jump! exploring search algorithms for simulating platformer players," *https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/9057/9023*, 2014.

[11] unity3d, "Unity3d," *https://unity3d.com/es/*, 2017.